O'REILLY®

TURING 图灵程序设计丛书

同构JavaScript应用开发

Building Isomorphic JavaScript Apps



[美] Jason Strimpel & Maxime Najim 著

张俊达 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍

张俊达

前端开发工程师，毕业于华南理工大学，《React快速上手开发》译者。现任职于唯品会，主要负责移动Web开发工作。关注前端领域的新技术，乐于分享。



图灵程序设计丛书

同构JavaScript应用开发

Building Isomorphic JavaScript Apps

[美] Jason Strimpel, Maxime Najim 著
张俊达 译

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

同构JavaScript应用开发 / (美) 杰森·史特林贝尔
(Jason Strimpel), (美) 马克西姆·纳吉姆
(Maxime Najim) 著 ; 张俊达译. -- 北京 : 人民邮电出
版社, 2017.10
（图灵程序设计丛书）
ISBN 978-7-115-46868-0

Ⅰ. ①同… Ⅱ. ①杰… ②马… ③张… Ⅲ. ①JAVA语
言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第223584号

内 容 提 要

本书将向你展示如何构建和维护属于自己的同构 JavaScript 应用。全书分为三部分，第一部分描绘不同种类的同构 JavaScript 的轮廓，第二部分介绍关键概念，第三部分提供业界同行的解决方案案例。通过阅读本书，你将了解到这种应用架构日益流行的原因，并能将其运用于解决关键的业务问题，如页面加载速度和 SEO 兼容性。

本书适合对同构 JavaScript 感兴趣的 Web 开发人员。

◆ 著　　　　[美] Jason Strimpel, Maxime Najim
　译　　　　张俊达
　责任编辑　朱　巍
　执行编辑　夏静文
　责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：11
　字数：260千字　　　　2017年 10 月第 1 版
　印数：1－3 000册　　　2017年 10 月北京第 1 次印刷
　著作权合同登记号　图字：01-2017-4856号

定价：49.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　构建第一个应用

第三部分 现实世界的解决方案

前言

Jason Strimpel

我在多年前就开始了 Web 开发的职业生涯，当时我在加州大学圣地亚哥分校担任行政助理。我的工作职责之一就是维护部门网站。那个年代的 Web 开发还是站点管理员、表格式布局和 CGI 程序的天下，且网景浏览器仍然是浏览器领域中的佼佼者。当时，我的专业知识还有很多不足，也缺乏经验。我清晰地记得，当时我很担心，如果我发送的电子邮件中存在拼写错误，收件人就会看到它们被红色下划线标记出来，正如我看到他们的拼写错误时一样。幸运的是，我的上司很耐心，他说我的邮件中并没有拼写错误，并向我传授了关于网站开发的许多要点。

转眼 15 年过去了，如今我在各大会议上发表演讲，管理开源项目，合著图书，成了这个领域的“专家”。有时候我会问自己：“我是如何取得今天的成就的？”这个问题的答案绝不仅仅是任由时间一天天地流逝——至少不完全是虚度时光。

为什么需要同构JavaScript

到目前为止，我的 Web 开发之路的最后一站是沃尔玛实验室的平台团队，我的同构 JavaScript 探索之旅就是从这里开始的。刚开始在沃尔玛工作时，我被分配到了一个负责开发 Web 新框架的团队。这个框架是从零开始开发的，其目标是支撑面向公众的大型网站。除了满足这类网站的最低要求，支持 SEO 并优化网页加载速度之外，保持 UI 工程师（包括我在内）的开发愉悦感和高效率也非常重要。很明显，为了更好地实现 UI 工程师的目标，我们的首选是基于现有的**单页面应用**（Single-Page Application，SPA，https://en.wikipedia.org/wiki/Single-page_application）技术进行扩展。但 SPA 模型有一个问题，它不能很好地支持我们的最低要求（详情请参见下文中的“完美风暴：一个极其平常的故事”和 1.2.3 节中的“单页面 Web 应用”），所以我们最终选择采用同构的方式。以下是这样做的理由。

- 对于同一个渲染周期，客户端和服务器端可以使用同一套代码。这意味着不需要重复劳动，可以在降低界面开发与维护成本的同时提高团队开发速度。
- 在服务器端渲染一份初始的 HTML 代码后，用户会感觉网页的加载速度更快，这是因为用户可以在浏览器中先看见首屏渲染的内容，而无须等待应用资源的加载和数据抓取完成。在网络延迟比较严重的环境中，这种改进页面预加载的方式尤为重要。
- 同构应用支持 SEO，因为同构应用使用的 URL 不包含 # 号片段。此外，在那些不支持 History API 的浏览器中，它可以（对后续的每次请求）优雅地降级为服务器端渲染的方式。
- 在支持 History API 的浏览器中，同构应用使用了 SPA 模型的分布式渲染，因此后续请求可以减轻服务器负载。
- 无论是在服务器端还是在客户端，UI 工程师都可以完全掌控界面（https://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/），而且同构应用在前后端之间划分了明确的界限，这有助于降低操作成本。

以上是我们团队走上同构 JavaScript 之路的主要原因。但在详细介绍同构 JavaScript 之前，先交代一些背景知识是很有必要的，这可以帮助你理解我们今天所处的具体环境。

平台的演进

Web 技术的快速演进令人难以想象。当初，CSS 和 JavaScript 技术被引入浏览器的目的是提供一种交互模型和关注点分离。你还记得曾有无数的文章提倡结构、样式和行为分离吗？即便在引入了这些技术之后，应用的架构也并没有发生太大变化。文档通过 URI 的形式进行请求，浏览器解析返回内容后，再进行渲染。唯一的不同之处就是 JavaScript 让界面变得更丰富了一点。直到微软公司引入一项被称为 `XMLHttpRequest`（https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest）的新技术，Web 才在这项技术的催化下演进为应用平台。

Ajax：应用平台的崛起

尽管很多前端工程师对微软公司及其 IE 浏览器嗤之以鼻，但他们仍然应当对微软怀有一份感激之情。如果没有微软，前端工程师未必能达到今天的职业高度。如果 `XMLHttpRequest` 技术没有出现，Ajax 技术也就不会诞生；如果没有 Ajax，就不会有这么多修改页面内容的需求，我们也就没有必要使用 jQuery（https://jquery.com/）。你猜接下来会发生什么呢？我们就不会有大量的前端 MV* 库可供选择，单页面应用的模式也不会出现，进而 History API 也不会出现。所以，下一次抱怨 IE 浏览器给你带来麻烦时，请你务必对微软作出客观评价，毕竟微软改变了历史进程，为今天的 Web 应用奠定了基础，还为你提供了一个锻炼思维的场所。

Ajax：技术债的积累

虽然 Ajax 技术影响了 Web 平台的发展进程，但它也以技术债的形式造成了一些破坏性的影响。Ajax 模糊了以前清晰定义的模型。过去，当用户导航到一个新页面或者提交表单数据时，浏览器只需依次发送请求、取得响应并解析完整的文档流。当 Ajax 成为 Web 开发的主流技术后，这种方式完全改变了。现在，工程师不需要向服务器请求重新获取整份文档，只需要根据用户的请求来判断是加载更多数据还是返回另一个视图。这意味着应用可以仅更新页面中的某个区域。这种特性大大优化了客户端和服务器端的性能，同时明显改善了用户体验。不幸的是，客户端的应用架构几乎变得不存在了，而本来负责处理视图层的那些工程师并没有应对这种范式转变的经验。日积月累，这些因素将应用的维护工作变成了噩梦。由于模型定义变得模糊，Web 开发从此经历了一段艰难成长的时期。

完美风暴：一个极其平常的故事

想象一下如下情景：你所在的团队负责维护一个商业应用的商品页面。在该页面的右侧有一个用于显示用户点评的轮播组件，用该组件可以翻页。当用户点击翻页按钮时，客户端会改变 URI 并向服务器发起请求以重新获取整个商品页面。这种低效的做法会让身为工程师的你感到非常苦恼。你认为没有必要刷新整个页面并调用一个重新渲染页面的数据请求，真正需要获取的只是下一页评论的 HTML 内容。好在你一直在跟踪行业内最新的技术发展，准备尝试使用最近学习的 Ajax 来解决这个问题。你收集了一些关于 Ajax 的概念证明，并向上司推荐了这项技术。这时候的你看起来就像巫师一样。像其他技术一样，这项技术最终会正式投入到生产环境中。

你对目前的成果感到非常满意，直到后来了解到一种新的数据交换格式。这种称为 JSON 的格式受到了 JavaScript 的非官方代言人 Douglas Crockford（http://www.crockford.com/）的极力推荐。你马上就觉得目前的实现不够完美了。第二天，你使用一种称为**微模板**（micro-templating，http://ejohn.org/blog/javascript-micro-templating/）的技术编写了一份新的概念证明，并再次向上司推荐。这项技术同样受到了好评并再次投入到生产环境中。

此时普通的工程师已经将你奉为神明。这时候，代码在审查阶段被发现有 bug。上司找到你并让你修复 bug，因为这段逻辑是你实现的。你审查了代码，并向上司宣称 bug 在服务器端的渲染中。然后你需要对使用两套渲染方案的原因进行一番解释。在解释完为什么 Java 不能在浏览器中运行后，你还得向上司保证这种实现是值得的，因为这样可以大大提升用户体验。这个问题像烫手的山芋那样被传来传去，直到最后才得以解决。

尽管违背了 DRY（don’t repeat yourself，不要重复你自己）原则，但你依然被誉为专家。你的实现模式逐渐被大家学习、模仿，进而充斥着整个代码库。但随着这种模式的渗透，意料之外的事情发生了。bug 的数量开始不断上升，开发人员开始害怕因修改代码而造成

的回归问题。目前欠下的技术债甚至比国家的财政赤字还要严重。工程经理和开发人员开始互相推卸责任。应用变得非常脆弱，公司也因此难以应对市场的快速变化。你感到一种强烈的罪恶感。幸好，你发现了一种叫作单页面应用的新模式……

客户端架构的救赎

近段时间，你阅读了一些文章，内容主要是人们对于前端架构的缺失而感到沮丧。这些人通常会将责任归咎于 jQuery，尽管这个库本来只是对 DOM 操作的封装（façade）。好在业界有人遇到了和你一样的困境，并且没有在其他不明真相的人的评论中停止自己的脚步。其中一个人就是 Backbone（http://backbonejs.org/）框架的作者 Jeremy Ashkenas（https://github.com/jashkenas）。

你开始了解 Backbone，阅读相关文章，并且深感兴趣。Backbone 将应用逻辑从数据检索中抽离出来，并将界面代码整合为单一语言和运行时，因此可以有效减少服务器端的压力。"找到了！"你在心中欢欣鼓舞地喊道。这个框架将解决我们遇到的所有问题。你又提出了一份新的概念证明，并开始实施。

在我们访问时发生了什么

你很快就被称为救世主。这种新的 SPA 模式在公司范围内被广泛接受。bug 的数量开始减少，工程师又重拾了信心。交付代码时的恐惧感几乎已经消失。这个时候，负责产品的同事找到你，并告知你自从实现 SPA 模式之后，网站的访问量下降了。你得想办法处理 # 号片段带来的问题了。经过一番详尽的研究后，你确定问题出在搜索引擎没有考虑 URI 中的 `window.location.hash` 部分，而 `Backbone.Router` 用这部分来创建可跳转、可收藏书签、可分享的页面视图。因此，当搜索引擎爬取这个应用时，没有任何可以收录的内容。现在你面临的形势更加严峻了，因为这个问题会对商品销量产生直接影响。因此，你再次开启了调研与开发的循环。结果你发现有两种方案可供选择：第一种方案是运转新的服务器，模拟 DOM 操作以运行客户端应用，并将搜索引擎重定向到这些新服务器上；第二种方案是付费让其他公司为你提供解决方案。除了 SPA 实现给公司带来的损失外，这两种方案都需要支付额外成本。

同构JavaScript：一个美好的新世界

上述这个故事综合了我的个人经历以及我从其他工程师那里目睹或听说的故事。如果你也曾为某个 Web 应用付出很多时间，我相信你也会有类似的经历和感受。故事当中的某些问题已经成为了历史，但部分问题依然存在。此外还有一些问题没有明说，例如页面加载速度有待优化以及缺少感知渲染。如果将路由的响应与渲染的生命周期合并为一个通用的代

码库，并同时支持在客户端和服务器端运行，应该就可以解决上述这些问题以及其他潜在问题。这就是同构 JavaScript 的意义所在。同构 JavaScript 应用是整合两种架构，以创建易于维护的、更好的用户体验。

未来的路

本书的主要目的是提供实现同构 JavaScript 所需的基础知识，帮助你理解业界现有的同构 JavaScript 解决方案。本书旨在提供足够多的信息，让你在实际中判断同构 JavaScript 是否为可行的解决方案，同时介绍业内最先进的解决方案，避免你重复造轮子。

第一部分是对这个主题的介绍。首先，详尽地介绍现有的几种 Web 应用架构，内容涵盖同构 JavaScript 的基本原理和用例，如 SEO 支持和提升页面的感知加载速度。然后，概述不同种类的同构 JavaScript 应用，如实时应用与类似 SPA 模式的应用。此外，还介绍了同构应用方案的组成部分，其中包括提供环境 shim 和抽象的实现，以及真正与环境无关的实现。该部分将为第二部分奠定代码基础。

第二部分将主题分解为关键概念，这些概念在大部分同构 JavaScript 解决方案中被普遍使用。每种概念都无须依赖现有的库［如 React（https://facebook.github.io/react/）、Backbone（https://facebook.github.io/react/）或 Ember（https://facebook.github.io/react/）］即可实现。这样做是为了避免将概念和某种特定的解决方案混淆。

第三部分将会介绍业内的专家是如何在他们的解决方案中作出权衡的。

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语或重点内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

代码示例

补充材料（代码示例、练习等）可以从 https://github.com/isomorphic-javascript-book 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Building Isomorphic JavaScript Apps* by Jason Strimpel and Maxime Najim (O'Reilly). Copyright 2016 Jason Strimpel and Maxime Najim, 978-1-491-93293-3"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice

Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网页地址是：

http://shop.oreilly.com/product/0636920042846.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

Jason Strimpel

首先，我要感谢妻子 Lasca 的耐心与支持。我每天都被你的智慧、幽默、慈悲和爱所包围。感谢你选择了和我一起生活，这是我莫大的荣幸。是你让我成为了更加优秀的人，感谢你对我的爱。

其次，我想感谢与我合著本书的同事 Maxime。没有你的激情、知识、想法和专业知识，我的努力将大打折扣。你对于软件架构的洞察力每天都能给予我灵感。非常感谢你。

感谢我的编辑 Allyson。你提出的观点、疑问和修改意见让我的写作增色不少。感谢你。

最后，我要感谢第三部分中所有内容的贡献者，感谢你们在百忙之中抽空和读者分享你们的故事。这一部分展示了同构 JavaScript 的深度以及解决方案的多样性。你们用独特的解决方案证明了每一位工程师都是富有创造力的。感谢你们。

Maxime Najim

首先要感谢我的家人——我的妻子 Nicole，以及我的孩子 Tatiana 和 Alexandra，感谢他们在本书的编写和出版过程中给予我的支持与鼓励。

我也永远感激 Jason 邀请我和他合著本书，能与你合作让我感到非常荣幸。这是一个千载难逢的机会，是你的智慧、知识和努力把这个机会变成了现实。我对你感激不尽。同样，非常感谢我们的编辑 Allyson 在整个过程中提供了宝贵的支持和建议。

最后，我要特别感谢第三部分内容的贡献者在百忙中抽空与我们分享他们的故事和经历。非常感谢你们！

电子书

扫描如下二维码，即可购买本书电子版。

第一部分

简介与关键概念

“黄金时代”一词最早源自早期的希腊和罗马诗人，而现在就成为了技术革新时段的代名词。在 20 世纪广播电视的黄金时代，作家和艺术家将他们的技能运用于新的媒介中，创造出了各种新鲜和引人注目的内容。或许我们现在就处于 JavaScript 的黄金时代，尽管这需要时间证明。但毫无疑问，JavaScript 已经为在浏览器中运行类似于桌面程序的应用这一新时代铺好了道路。

在过去的十年中，Web 已经演进为一个构建丰富、高度交互应用的平台。浏览器不再仅仅是一个文档渲染工具，Web 也不再仅仅是将各种文档链接到一起。**网站**已经发展为**Web 应用**。这意味着越来越多的 Web 应用逻辑将会在浏览器端，而非服务器端运行。然而，在过去的十年中，用户的期望也在不断提高，首屏加载速度显得越发重要。Radware 的一份报告（http://www.slideshare.net/Radware/2015-spring-state-of-the-union-ecommerce-page-speed-web-performance-infographic）显示，1999 年，一般用户愿意等待的页面加载时间为 8 秒。到 2010 年，57% 的在线购物者称，如果一个页面在 3 秒之内没有显示任何内容，那么他们就会将它关闭。调查结果正好反映出了 JavaScript 黄金时代的问题所在：客户端 JavaScript 在丰富页面内容和增强交互性的同时，也延长了页面加载时间，这使得用户在初次加载时的体验非常糟糕。页面加载时间最终会影响公司的效益。amazon.com 和 walmart.com 的报告都显示：页面加载时间每缩短 100 毫秒，他们的收入就会获得高达 1% 的增长（http://www.globaldots.com/how-website-speed-affects-conversion-rates/）。

我们将在本书的第一部分中介绍同构 JavaScript 的概念，以及同构渲染如何大幅提升用户体验。我们还将以图谱的形式探讨同构 JavaScript，简述几种不同类型的同构代码。最后，我们将目光放远，看看同构 JavaScript 如何基于服务器端渲染技术，创建出复杂、实时更新、支持协作的实时应用。

第 1 章

为什么需要同构JavaScript

Jason Strimpel、Maxime Najim

2010 年，Twitter 对其网站进行了一次重构，并发布了新的版本。这个称为“#NewTwitter”的新版本将 UI 渲染和业务逻辑放在了 JavaScript 中，并在用户的浏览器中运行。这种架构在当时是开创性的。然而，不到两年的时间，Twitter 再次进行了重构，将渲染功能移回了服务器端。Twitter 的这次改版将页面的初始渲染时间缩短到了原来的五分之一（https://blog.twitter.com/2012/improving-performance-on-twittercom）。Twitter 的做法在 JavaScript 社区中引起了轰动。开发者和其他许多人很快意识到，客户端渲染对性能有着非常明显的影响。

构建客户端 Web 应用的最大劣势在于，首次加载需要付出高昂的代价下载一个 JavaScript 大文件。互联网中的主要传输协议是 TCP（Transmission Control Protocol，传输控制协议），该协议定义了一种被称为**慢启动**（slow start）的拥塞控制机制，这意味着数据是以逐渐增加数据块的方式进行发送的。Ilya Grigorik 在《Web 性能权威指南》[1] 一书中解释了 TCP 协议如何经过“客户端与服务器端之间的 4 次往返……以及几百毫秒的延迟，才能达到 64KB 的吞吐量”。显然，发送给用户的前几千字节的数据对良好的用户体验和页面响应性至关重要。

客户端 JavaScript 应用在初始化时只包含一个 <script> 标签和一个空的 <body> 标签，这类应用的崛起产生了一些问题：初始化加载速度慢、需要对 URL 进行 hashbang（#!）的特

注 1：此书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1194。—— 编者注

殊处理（随后将对此进行详细介绍），以及糟糕的搜索引擎检索性。通过将客户端和服务器端代码合二为一，同构 JavaScript 解决了这些问题。同构 JavaScript 提供了整合两种架构的能力，可以创建易于维护的、用户体验良好的应用。

1.1　定义同构JavaScript

简单来说，同构 JavaScript 应用就是在浏览器客户端和 Web 应用服务器端间共享同一套 JavaScript 代码的应用。从某种意义上讲，之所以称为同构，是因为无论在客户端还是在服务器端运行，应用都具有相同的形式或形态。同构 JavaScript 是 JavaScript 发展进程中的革命性一步。但就像钟摆一样，软件开发中的进步通常不稳定，来来回回。如果从事软件开发已有一段时间，那么你可能已经了解过一些时隐时现的设计方法。在某些情况下，我们似乎永远无法找到正确的平衡点。

近 20 年来，Web 应用的发展方式非常符合这一规律。我们见证了 Web 的演进——从最初简陋的蓝色超链接静态页到如今用户体验丰富、可以媲美成熟原生应用的平台。之所以能做到这一点，是因为 Web 的客户端－服务器模型迅速从**重服务器端、轻客户端**的方式转变为**轻服务器端、重客户端**的方式。然而，这种方式的转变导致了大量问题，我们将在本章后面具体讨论。就目前而言，可以简单地概括为我们需要在**重客户端**和**重服务器端**之间取得平衡。为了真正了解这种平衡的意义，我们必须先后退一步，看看 Web 应用在过去几十年里是如何发展的。

1.2　评价其他的Web应用架构方案

要想理解同构 JavaScript 方案的由来，必须先了解这个方案出现时的状况。首先要确认主要的使用场景。

> 第 2 章中介绍了两种类型的同构 JavaScript 应用并分析了其架构。本书探讨的同构 JavaScript 应用场景是电子商务相关的 Web 应用。

1.2.1　状况的改变

万维网（World Wide Web）的出现要归功于 Tim Berners Lee（https://www.w3.org/People/Berners-Lee/）。当时他在一个核研究机构中工作，并在一个名为 Enquire（https://en.wikipedia.org/wiki/ENQUIRE）的项目中尝试使用了超链接技术。1989 年，Tim 整理了超链接的概念，提议在一个提供文档链接的中央数据库中应用这项技术。随着时间推移，数据库变得越发庞大，对我们的日常生活（如通过社交媒体）和商业（电子商务）产生了巨大影响。我们这些青少年都深陷到这个虚拟的大商场中。丰富多样的内容和购物选项能够帮助我们在购买时

作出明智的决定。在意识到消费者的选择多如牛毛后，企业非常关心我们能否找到并查看他们的内容与商品，因为他们的最终目标是提高转换率（让我们购物）。为此，甚至还出现了专门负责搜索引擎优化（search engine optimization，SEO）的专家，这些专家唯一的工作就是让企业的内容和商品出现在搜索结果前列。然而，有关转换率的斗争并没有到此结束。一旦消费者找到商品，页面必须能够快速加载并响应用户的交互操作，否则企业可能会将消费者拱手让给竞争对手。这正是我们身为工程师应当发挥作用的地方，而除了企业的关注点外，我们还有自己的一系列关注点。

1.2.2 工程上的关注点

作为工程师，我们也有自己的一些担忧，主要关于可维护性和效率，但这并不是说我们在权衡技术决策时不会考虑企业的关注点。事实上，优秀工程师的做法恰恰相反：他们会基于手头的业务问题权衡短期和长期的利弊，为每个可能发生的业务问题寻找最优解。

1.2.3 可选架构

考虑到我们的主要业务场景是电商应用，我们来看看在 Web 发展史中适用于该场景的几种架构。但在此之前，我们先要明确一些关键的评判标准，以便公正地评估不同的架构。以下标准是按照重要性排序的。

(1) 应用要能够收录在搜索引擎中。
(2) 应用的首屏加载速度应该是优化过的，也就是说，**关键渲染路径**（critical rendering path）应该属于初始响应的一部分。
(3) 应用要能够响应用户的交互操作（比如优化后的网页切换）。

> 关键渲染路径指的是页面上与用户的主要操作相关的内容。在电子商务应用中，关键渲染路径是对商品的描述。对新闻网站来说，关键渲染路径则是一篇文章的内容。

在整个评估过程中，必须权衡这些业务标准和工程的主要关注点（可维护性和效率）。

1. 传统的Web应用

前面提到过，设计和创造 Web 的最初目的是共享信息。由于万维网的提出是以 Enquire 项目的成功作为前提的，因此 Web 在起步阶段仅仅用于多页文档的相互连接不足为奇。20 世纪 90 年代初，大部分的 Web 内容都是以完整的 HTML 页面的形式渲染的。当时支持这种方式的机制是 HTML、URI 和 HTTP（现在也依然如此）。HTML（Hypertext Markup Language，超文本标记语言）是一种标记规范，当浏览器解析标记时，会将其转换为文档对象模型。URI（Uniform Resource Identifier，统一资源标识符）用于标识资源的名称，即

应该响应请求的服务器的名称。HTTP（Hypertext Transfer Protocol，超文本传输协议）是负责连接一切的传输协议。这三种机制为互联网提供了动力，并形成了传统 Web 应用的架构。

传统的 Web 应用指的是：所有的标记——至少是关键渲染路径的标记——是通过服务器使用某种服务器端语言（如 PHP、Ruby、Java 等）进行渲染的，如图 1-1 所示。浏览器解析文档后，用于丰富用户体验的 JavaScript 代码会被初始化。

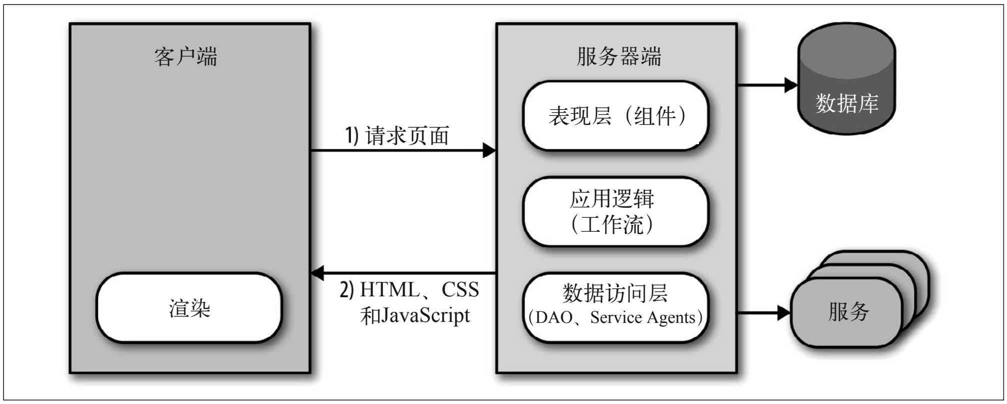
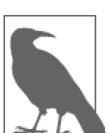


图 1-1：传统 Web 应用的流程

简而言之，上图展示了传统 Web 应用的架构。我们来研究一下这个架构是否符合我们的评估标准和工程上的关注点。

首先，它很容易被搜索引擎收录，因为当爬虫遍历应用时，所有的内容都是可爬取的，所以消费者是可以搜索到应用内容的。其次，页面加载也经过了优化，因为关键渲染路径的标记是通过服务器端进行渲染的，从而提高了感知的渲染速度，降低了用户跳出应用的可能性。然而，传统的 Web 应用只能满足上述三点要求中的两点。



我们所说的"感知的渲染速度"是什么意思呢？ Ilya Grigorik 在《Web 性能权威指南》一书中是这样解释的："时间测量是客观的，而时间感知是主观的。我们可以通过设计来改善感知性能。"

在传统的 Web 应用中，页面导航和数据传输都遵循 Web 原本设计的方式进行。当用户导航到一个新页面或者提交表单数据时，浏览器会发送请求、取得响应并解析完整的文档流，即使页面中只有部分信息改变了。这种方式在实现前两个评判标准时极为有效，但这种全页面安装和拆卸的生命周期的代价非常高，因此在响应性方面这只是一个次优解。因为有幸生活在拥有 Ajax 的年代，所以我们都已经知道还有比整页刷新更加高效的方法。但引入 Ajax 也会带来成本，我们将会在下一节中具体探讨。但在进入下一节之前，我们

应该先看看在传统 Web 应用的背景下是如何应用 Ajax 的。

Ajax 时代。星星之火可以燎原，而 `XMLHttpRequest`（https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest）正是点亮 Web 平台的星火。然而，这项技术在集成到传统 Web 应用中时并没有给人留下太深刻的印象，这并不是因为设计或者技术本身的原因，而是因为负责集成该技术到传统 Web 应用中的那些人缺乏使用经验。在大多数情况下，负责该工作的人都是刚开始专攻视图层的设计师。我自己是从行政助理转为设计师和开发者的。当时，我的这两项技能都不足。不用说，我对过去参与过的应用造成了很大的破坏（不过我认为这是我对平台演进的贡献）。不幸的是，在这段演进期，我和那些缺乏适当培训与指导的人们接触的所有应用都受尽了苦头——它们的过程重复、关注点混乱。有一个很好的例子可以突出这些问题，即相关商品的轮播组件（如图 1-2 所示）。

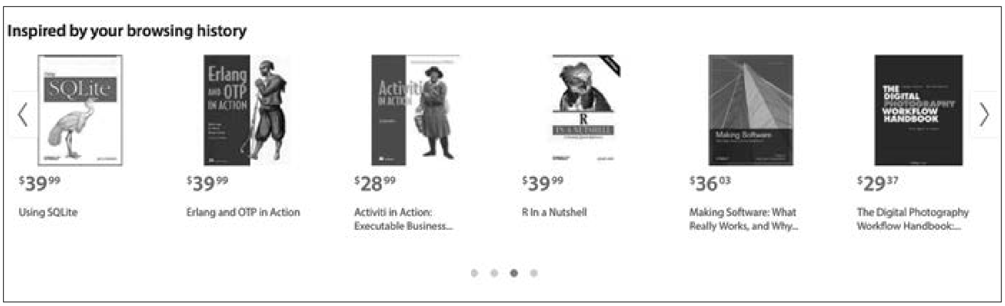


图 1-2：商品轮播组件示例

（相关）商品轮播组件可以分页浏览产品。在某些情况下，所有产品都是预先加载的，但有时会因为商品数量太多而不能采用预加载。在第二种情况下，需要发起网络请求以获取下一页的商品信息。由于刷新整个页面的效率极低，因此典型的解决方案是在翻页时使用 Ajax 获取新的商品页面集。接下来可优化的是，只获取渲染页面集所需要的数据，这意味着你需要创建用于重复生成的模板、模型和静态资源，并在客户端进行渲染（如图 1-3 所示）。这种做法需要编写更多单元测试。这个例子非常简单，但如果将这种思想推广到大型应用中，你将发现应用会变得难以跟踪与维护——你不能轻易地推断出应用是如何结束在某个特定状态的。此外，重复编写渲染逻辑是一种资源浪费，而且在添加或者修改功能时，同时操作两份 UI 代码会导致应用出现 bug 的概率增高。

由于启用了 Ajax，再加上看似美好的初衷，导致了 UI/ 视图层的分裂与复制，一个看似精心构造的应用就此化作瓦砾，从而让无数工程师遭受了挫折。好在工程师在沮丧的时候通常是最有创造力的。正是这种挫折推动了创新，再结合工程师扎实的工程技能，便造就了下一代应用架构。

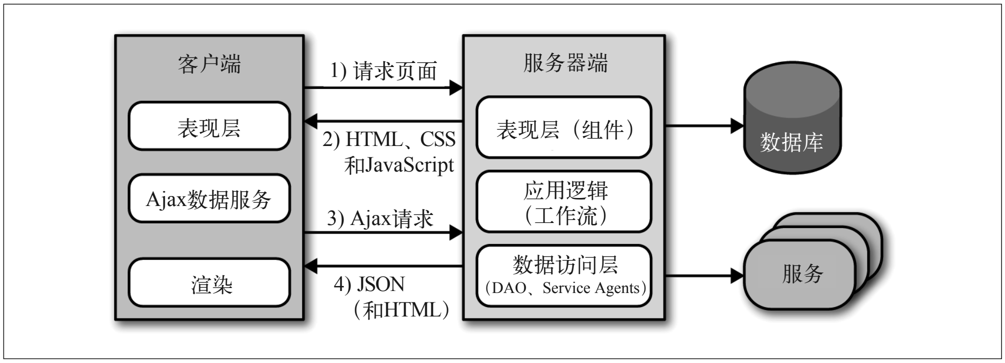


图 1-3：使用 Ajax 的传统 Web 应用流程

2. 单页面Web应用

一切事物都有自己的循环周期。在 Web 开始阶段时流行的轻客户端可能给了 Sun Microsystems 的 NetWork Terminal（NeWT，https://en.wikipedia.org/wiki/Sun_Ray）以启发。但到了 2011 年，Web 应用开始放弃轻客户端模型，并过渡到重客户端模型，而操作系统领域在多年以前就发生过这样的变化。巨石已经浮出水面。这就是单页面应用架构的黎明。

通过将渲染工作完全转移到客户端来进行，SPA 解决了一直以来困扰着传统 Web 应用的问题。该模型将应用逻辑从数据检索中抽离出来，并将 UI 代码整合为单一语言和运行时，因此可以有效地减少服务器端的压力（如图 1-4 所示）。

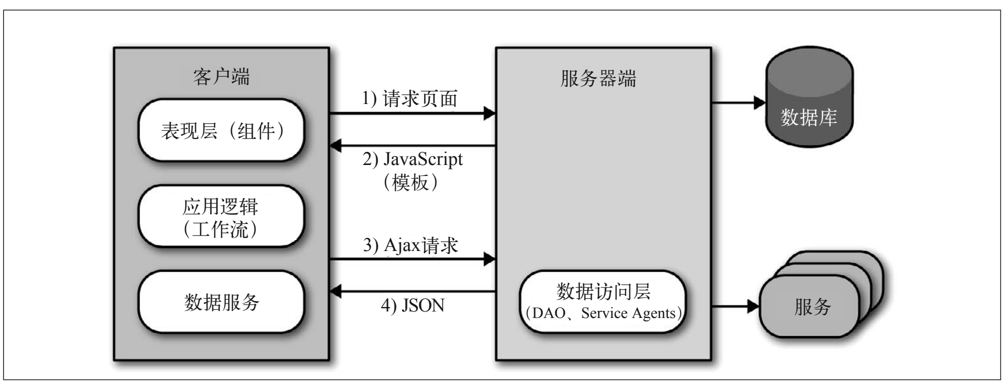


图 1-4：单页面应用的流程

之所以能减少服务器端的压力，是因为服务器先将一份包含了静态资源、JavaScript 和模板的静荷数据（payload）发送到了客户端。之后，客户端只需要获取渲染页面或视图所需要的数据即可。这种行为显著提高了页面的渲染效果，因为避免了在用户请求新页面或提

交数据时重新请求并解析页面的性能开销。除了性能收益外，这种模型还解决了将 Ajax 引入传统 Web 应用中所产生的工程问题。

现在回到之前讨论的产品轮播组件示例，第一页（相关）产品的信息在以前是由应用服务器渲染的。在翻页时，客户端负责发起随后的请求并进行渲染。在现代 Web 平台中，这种职责的模糊界限和工作的重叠正是传统 Web 应用面临的主要问题。但这些问题在 SPA 中将不复存在。

在 SPA 中，服务器端和客户端之间存在明确的界限。API 服务器响应数据请求，应用服务器提供静态资源，而客户端则负责展示。在这个产品轮播的例子中，应用服务器会向浏览器发送一份仅包含 JavaScript 静荷数据和模板资源的空文档。客户端应用在浏览器中进行初始化并向服务器请求渲染视图所需要的数据，视图中包含了轮播组件。收到数据后，客户端应用将会为这次轮播渲染产品的第一组集合。在翻页时，请求数据和渲染的生命周期会再次重复，并且复用同一段代码路径。这确实是一种优秀的工程解决方案，但问题是，这种方案并不能在任何时候都提供最佳的用户体验。

在 SPA 中，终端用户感知到的初始页面加载速度可能会非常缓慢，因为用户必须等到数据请求完成才能看见页面渲染。因此，在页面加载时，用户最多只能看到加载指示器动画，而不能立即看到内容。针对渲染延迟的问题，一种常见的折中方案是，为初始页面的数据提供专门的数据优化服务。但这样做就需要编写额外的服务器端应用逻辑，从而导致两端职责范围再次变得模糊，还需要额外维护另外一层代码。

SPA 面临的第二个问题关系到用户体验和企业利益。在默认情况下，SPA 对 SEO 不友好，这意味着用户不能通过搜索引擎找到与应用相关的内容。这个问题源于 SPA 利用了 hash 片段实现路由。在分析这种方式为什么会影响 SEO 之前，我们先看看 SPA 路由的机制。

SPA 依赖 hash 片段将人造的 URI 路径映射到路由处理器中，该处理器会渲染对应的视图。举个例子，在传统的 Web 应用中，“关于我们”的页面 URI 可能是 http://domain.com/about，但在 SPA 中则可能是 http://domain.com/#about。SPA 在 URL 的末尾添加了一个 # 号和一个片段标识符。SPA 路由之所以要利用 hash 片段，是因为片段的内容发生变化时，浏览器不会像 URI 发生变化时那样发起新的网络请求。这一点至关重要，因为 SPA 的整个大前提就是只请求页面或视图渲染所需要的数据，而不是为每一个页面获取并解析整份文档。

SPA 片段对 SEO 不友好的原因是，hash 片段不会作为 HTTP 请求中的一部分发送给服务器（按照规范定义）。对于 Web 爬虫而言，http://domain.com/#about 和 http://domain.com/#faqs 是同一个页面。好在谷歌定义了一种变通方案，为 hash 片段提供了 SEO 支持，这个方案就是使用“#!”（hashbang）。

大多数的 SPA 库目前已经支持 History API（https://developer.mozilla.org/en-US/docs/Web/API/History），并且谷歌的爬虫最近对于索引 JavaScript 应用提供了更好的支持——在此之前，JavaScript 代码甚至不会被 Web 爬虫所执行。

按照谷歌的规定，其基本前提是将 SPA 路由片段中的“#”替换为“#!”，因此 http://domain.com/#about 需要更改为 http://domain.com/#!about。这样一来，谷歌的爬虫才能确定这个页面的内容需要被索引，而不仅仅是简单地锚点。

锚点标签用于在文档内部创建内容链接。

随后，爬虫将这个链接转换为完全合格的 URI 版本，因此 http://domain.com/#!about 会变成 http://domain.com/?query&_escaped_fragment=about。然后，服务器端负责将 SPA 对应的屏幕快照提供给爬虫。图 1-5 展示了该过程。

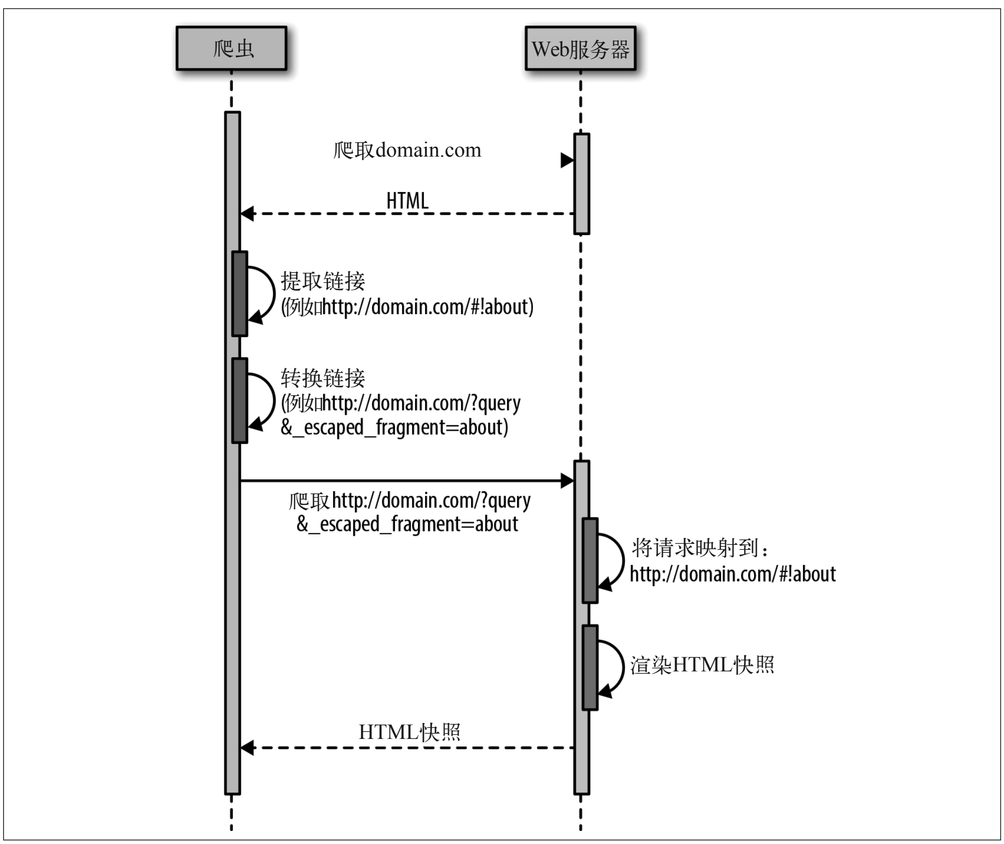


图 1-5：爬虫收录 SPA URI 的过程

此时，SPA 的价值主张愈发下降了。从工程角度来说，需要在下列方案中二选一。

(1) 在服务器中运行一个无界面的浏览器，如 PhantomJS（http://phantomjs.org/），用于在服务器中运行 SPA 并响应爬虫请求。
(2) 将这个问题外包给第三方供应商解决，如 BromBone（http://www.brombone.com/）。

这两种修复方案都需要成本，而且还不包括前面提及的首屏渲染不理想的成本。好在工程师都善于解决问题。正如从传统 Web 应用到 SPA 的改进，新一代架构诞生了，也就是同构 JavaScript。

3. 同构JavaScript应用

同构 JavaScript 应用是传统 Web 应用和 SPA 架构的完美结合。同构应用具备以下优势。

- SEO 默认支持使用完全合法的 URL——不再需要“#!”的变通方案了——通过 History API 进行跳转，在不支持 History API 的浏览器中可以优雅地回退到服务器端渲染模式。
- 在支持 History API 的浏览器中，后续的页面请求使用了 SPA 模型的分布式渲染。这种实现还可以减轻服务器的负载。
- 对于同一个渲染周期，客户端和服务器端可以重用同一套代码。这意味着我们不需要重复劳动，也不会让界限变得模糊。这可以在降低 UI 开发成本与 bug 数量的同时，提高团队的开发速度。
- 通过在服务器端渲染首屏页面提高加载速度。用户不再需要在首屏渲染之前等待网络请求完成和一直看着加载指示器动画了。
- 纯 JavaScript 技术栈，这意味着应用界面的代码（https://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/）可以由前端工程师单独维护，而无须经过后端工程师。关注点和责任更清晰地分离，这使得每个人都可以只在自己擅长的领域贡献代码，从而做到术业有专攻。

同构 JavaScript 架构可以同时满足本节前面提到的三个评判标准。同构 JavaScript 应用可以轻松地被所有的搜索引擎收录，并能优化网页加载速度和页面之间的过渡（适用于支持 History API 的浏览器，而在老版本浏览器中可以优雅地降级，不会对应用架构产生影响）。

1.3　附加说明：何时不使用同构

像 Yahoo!、Facebook、Netflix 和 Airbnb 这些公司已经接受了同构 JavaScript。然而，同构 JavaScript 架构可能仅仅适用于某些类型的应用。正如我们将在本书中探索的那样，同构 JavaScript 应用需要更多架构上的考虑，实现上也存在一定的复杂度。对于 SPA 来说，如果性能要求不高或者没有 SEO 需求（比如需要登录后才能使用），同构 JavaScript 带来的麻烦似乎远大于收益。

此外，很多公司和组织可能还没准备在服务器上操作和维护一个 JavaScript 的执行引擎。例如，大量使用 Java、Ruby、Python、PHP 的组织可能并不知道如何在生产环境中对一个 JavaScript 应用服务器（如 Node.js）进行监控与故障诊断。在这些情况下，同构 JavaScript 可能会引起难以承受的额外操作成本。

> Node.js 提供了一个出色的服务器端 JavaScript 运行时。对于使用了 Java、Ruby、Python 或者 PHP 的服务器来说，有两种主要的候选方案：一是在正常的服务器之外再运行一个 Node.js 进程，将后者作为本地或者远程的“渲染服务”；二是使用嵌入式 JavaScript 引擎（比如集成在 Java 8 中的 Nashorn）。然而，这两种方案都有明显的缺点。运行 Node.js 作为渲染服务需要在进行 socket 通信时序列化数据，这带来了额外的开销。同样，在其他语言中使用的嵌入式 JavaScript 引擎通常是没有经过优化的，可能会导致额外的性能问题（尽管这会随着时间的推移得到改善）。

如果你的项目或者公司不需要借助同构 JavaScript 架构提供的便利（如本章所述），请务必针对具体工作选择合适的技术。然而，当服务器端渲染不在你的选择范围之内，而你又需要关注首屏加载速度和搜索引擎优化时，别担心，本书可以帮到你。

1.4　小结

我们在本章中定义了同构 JavaScript 应用——在浏览器客户端以及 Web 应用服务器端共享同一套 JavaScript 代码的应用——并确定了本书中主要讨论的同构 JavaScript 应用类型是电商应用。随后，我们回顾了 Web 的发展历史并研究了其他架构的发展历程，通过 SEO 支持、首屏加载速度优化和页面过渡效果优化这三个关键的验收标准评估了这些架构。我们看到，同构 JavaScript 出现之前的架构不能满足所有的验收标准。在本章最后，我们将传统 Web 应用和 SPA 的优势结合起来，得到了同构 JavaScript 应用架构。

第 2 章

同构JavaScript图谱

Maxime Najim

根据客户端和服务器端代码共享程度的不同，可以将同构 JavaScript 划分出一个图谱（如图 2-1 所示）。在图谱的左侧，客户端和服务器端共用最低限度的视图渲染（如 Handlebars.js 的模板），以及某些名字、日期或 URL 的格式化代码，或者是应用逻辑的某些部分。在图谱的这一侧，我们通常会发现客户端和服务器端通过共用模板的方式共享视图层，并共用辅助函数（如图 2-2 所示）。这些应用需要的抽象程度不高，因为在 JavaScript 中已经有一些流行的库支持在客户端和服务器端之间共用代码，比如 Underscore.js（http://underscorejs.org/）和 Lodash.js（https://lodash.com/）。

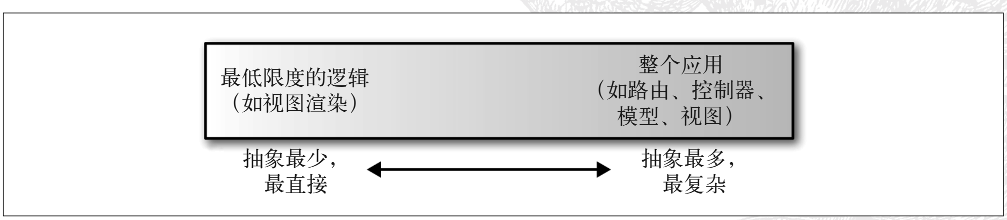


图 2-1：同构 JavaScript 图谱

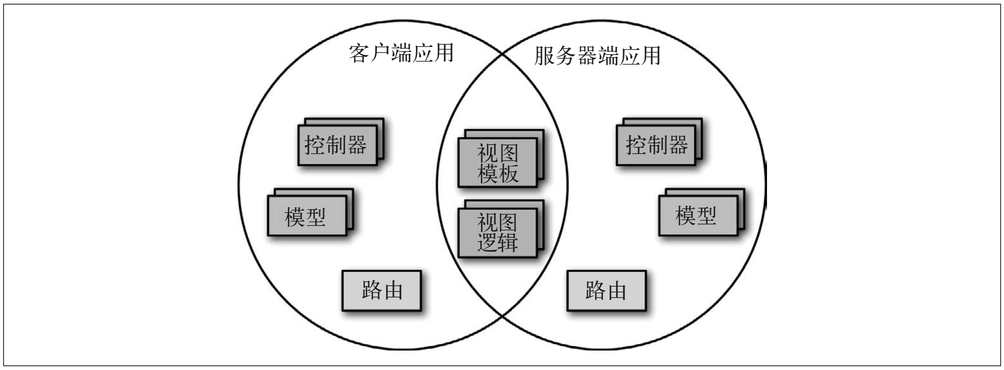


图 2-2：共享视图层

在图谱的右侧，客户端和服务器端共享整个应用（如图 2-3 所示）。共享内容包括整个视图层、应用程序流、用户访问限制、表单验证、路由逻辑、模型和状态。这些应用需要的抽象程度更高，因为客户端代码的执行上下文是 DOM（document object model，文档对象模型）和 window，而服务器端代码的执行上下文是一个请求 / 响应对象。

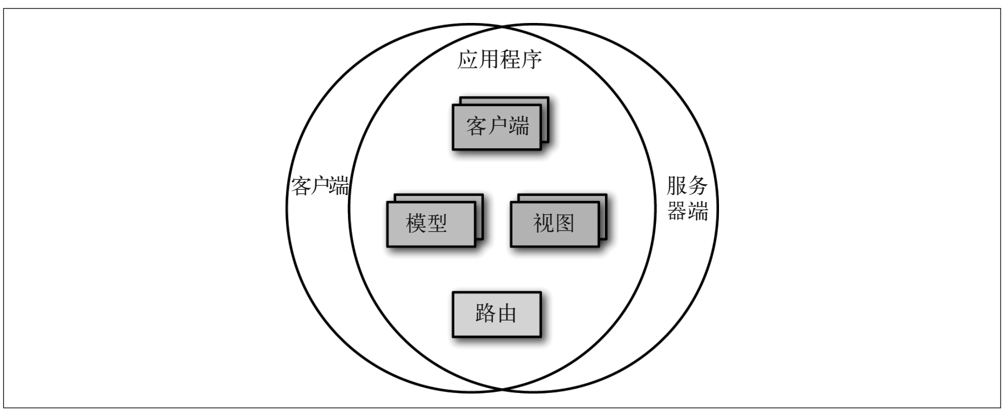
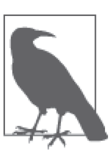


图 2-3：共享整个应用



在本书的第二部分中，我们将深入了解客户端和服务器端的代码共享机制。但本章将通过介绍在客户端和服务器端共享代码的功能不同的几种应用，简要地探索图谱两端的机制差异。我们也将明确指出需要的某些抽象，以便这些功能同构地工作。

2.1　共享视图

通过减少用户在跳转页面时重新加载完整页面的次数，并在用户交互时采用渲染页面部分内容的方式，SPA 提供了更加流畅的用户体验。SPA 利用客户端模板引擎技术来

接收模板（模板中包含了一些简单的变量占位符）、传递模型上下文对象、执行并输出 HTML，最终将结果插入到 DOM 树中。客户端模板将视图标记从视图逻辑中分离出来，从而创建更加易于维护的代码。而同构应用中的共享视图意味着模板和相对应的视图逻辑都需要共享。

2.1.1　共享模板

为了获得更快的（感知）性能和更佳的 SEO 效果，我们希望服务器端像客户端一样能够渲染任意的视图。在客户端，模板渲染很简单，只需要对一个模板进行求值，并将输出插入到某个 DOM 元素即可。但在服务器端，同一个模板会被渲染成字符串并在响应中作为结果返回。同构视图渲染的棘手之处在于，客户端需要接手完成服务器端未完成的事情。这通常称为**从客户端到服务器端的过渡**（client/server transition），也就是说，应用在浏览器中加载完成后，客户端需要进行适当的转换，以免“破坏”服务器端生成的 DOM 字符串。服务器需要将应用状态提取到一个对象中（称为 dehydrate），并发送给客户端，随后客户端需要使用同一状态初始化应用，并将视图还原（rehydrate）为与在服务器上相同的状态。

例 2-1 展示了一种典型的服务器响应，即在页面的 body 部分渲染某些标记，并且使用 `<script>` 标签输出经过序列化的状态。服务器将序列化状态放入渲染的视图中，客户端需要对状态进行反序列化，并将状态和预先渲染的标记关联起来。

例 2-1　引入服务器端渲染的标记与状态

```
<html>
  <body>
    <div>[[server_side_rendered_markup]]</div>
    <script>window.__state__=[[serialized_state]]</script>
    ...
  </body>
</html>
```

2.1.2　共享视图逻辑

模板的 helper 是对象，如数字、字符串或散列对象，通常比较容易共享。对于日期这样的格式化数据的共享，许多格式化库都同时支持在服务器端和客户端运行，比如 Moment.js 可以同时在服务器端和客户端进行日期的解析、验证、操作与显示。另外，URL 的格式化需要在路径前面添加主机名和端口，而在客户端中只需要简单地使用相对 URL 即可。

2.2　共享路由

大部分现代的 SPA 框架都支持路由的概念，路由负责在用户跳转页面时跟踪用户的状态。在 SPA 中，路由是控制跳转事件、改变状态和页面视图，以及更新浏览器跳转历史的主要机制。在同构应用中，我们同样需要一套路由配置（即将 URI 的模式映射到路由处理器

中），而且这套配置能够在服务器端和客户端之间方便地进行共享。共享路由的难点在于路由处理器自身，因其经常需要访问与环境相关的 API 来获取 URL 信息、HTTP 头部和 cookie 等。在服务器端，这些信息可以通过请求对象的 API 取得，而在客户端则需要通过浏览器的 API 取得。

2.3 共享模型

模型通常被称为业务对象、域对象或者实体。通过移除状态存储并从 DOM 恢复，模型为数据建立了一种抽象。在最简单的实现中，同构应用可以使用服务器端返回首屏响应之前一模一样的状态，对客户端应用进行初始化。在同构 JavaScript 图谱的一个极端中，服务器端和客户端共享状态与模型的定义规范包括双向同步（第 4 章将对这种实现进行更为详细的探讨）。

2.4 小结

应用在同构 JavaScript 图谱中可以处于不同的位置。客户端和服务器端共用的代码量不尽相同，从共享模板到共享应用的整个视图层，再到共享应用的大部分逻辑。随着在同构 JavaScript 图谱中的位置不同，应用可能需要建立更多的抽象。在下一章中，我们将会讨论不同类别的同构 JavaScript，并且更深入地分析这些抽象。

第 3 章

同构JavaScript分类

Maxime Najim

同构 JavaScript（isomorphic JavaScript）这一术语公认的出处是 Charlie Robbins 在 2011 年发表的博文“Scaling Isomorphic Javascript Code”（https://blog.nodejitsu.com/scaling-isomorphic-javascript-code/）。随后，这个术语在 Spike Brehm 于 2013 年发表的博文“Isomorphic JavaScript: The Future of Web Apps”（http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/）及随后的一些文章和会议演讲中多次出现，并因此开始流行起来。然而，在 JavaScript 社区（https://www.oreilly.com/ideas/renaming-isomorphic-javascript），关于同构的用词“isomorphic”曾存在一些争论。Michael Jackson（React.js 讲师、`react-router` 项目作者之一）认为，应该将“同构 JavaScript”称为“universal JavaScript”（https://medium.com/@mjackson/universal-javascript-4761051b7ae9#.h655sp39b）。Jackson 认为 universal 这个词可以突出“JavaScript 代码不仅可以在服务器端和客户端上运行，还可以在原生设备和嵌入式架构上运行”的特点。

而另一方面，isomorphism 是一个数学术语：对于两个数学对象来说，如果我们简单地忽略它们的个体差异，则当它们具有相似的属性和操作时，就是同构的。当我们将这个概念应用到图论中时，一切就变得很好理解了。图 3-1 中的这两个图就是很好的例子。

尽管这两个图看起来差别很大，但它们却是同构的。这两个图具有相同的结点数，而且每个结点拥有相同的边数。但它们是同构图的真正原因是，左图中的每个结点都能映射到右图中对应的结点，并且同时保留某些属性。比如，结点 A 可以映射到结点 1，并且右图中结点 1 的相邻关系和结点 A 是一致的。事实上，左图中的每个结点映射到右图后都保留了原有的相邻关系。

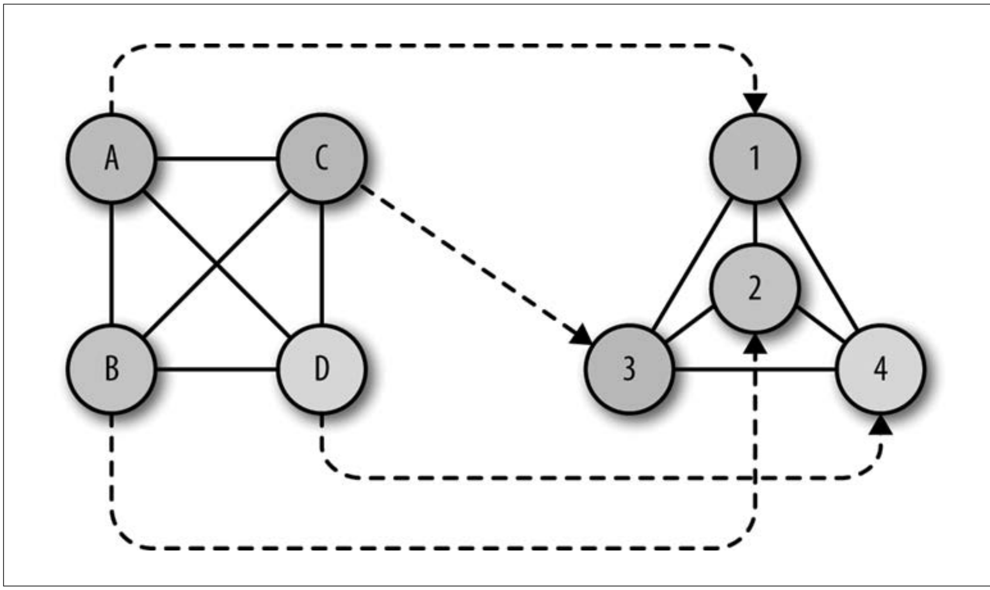


图 3-1：同构的图

这就是“同构”这个类比有意思的地方。要想让 JavaScript 代码可以同时在客户端和服务器端环境中运行，这些环境就必须是同构的；这也就是说，应该存在一种映射，能够将客户端的功能映射到服务器端的环境中，反之亦然。正如图 3-1 中的两个同构图中的映射关系那样，同构 JavaScript 环境也需要有映射关系。

在 JavaScript 中，不依赖于特定环境属性的代码可以轻松地在不同的环境中同时运行，比如那些避免使用 `window` 和 `request` 对象的代码。但对于 `req.path` 和 `window.location.pathname` 这样使用了特定环境属性的代码，则需要提供一种映射关系（有时被称为 shim）来抽象或“填充”到某个特定环境属性中。这使得同构 JavaScript 分成了两大类：与环境无关的，和为每个特定环境提供 shim 的。

命名与分类

同构 JavaScript 是一个不断演化的主题，其命名与分类也在定型的过程中。一般来说，应用代码可以分为两种类别：使用了环境 API（如 `window` 对象的 API）的代码以及无须使用特定环境 API 的代码，后者无须额外修改即可“到处”运行。针对使用了环境 API 的代码，我们一般有两种方案：一是修改环境，让代码可以同时在浏览器端和服务器端运行；二是根据环境 API 创建一个抽象层，并在应用代码中使用这些抽象方法。这两者的区别很小，还有人建议再添加第三种类别，即第二种做法外加“保留原有语义”。在本章中，我们将这两种方案统称为“为每个特定环境提供 shim”。关于命名的进一步讨论，请参见 16.1.2 节。

3.1　与环境无关的代码

与环境无关的 Node 模块只能使用纯 JavaScript 的功能，并且不能使用环境特定的 API 或者 `window`（浏览器端）和 `process`（服务器端）这样的属性。例如，Lodash.js、Async.js、Moment.js、Numeral.js、Math.js 和 Handlebars.js 都是与环境无关的。事实上，很多模块都属于这一类别，且这些模块都能够很好地在同构应用中运行。

唯一需要解决的问题是，这些 Node 模块是使用 Node 环境中的 `require(module_id)` 模块装载器进行加载的，但浏览器本身不支持 Node 环境中的 `require(...)` 方法。要想处理这个问题，我们需要一个负责在浏览器中编译 Node 模块的构建工具。目前有两个主流的构建工具可以完成这项工作，它们分别被称为 Browserify 和 Webpack。

在例 3-1 中，我们使用 Moment.js 定义了一个日期格式化方法，这个方法将同时在服务器端和客户端运行。

例 3-1　定义一个同构的日期格式化函数

```
'use strict';

var moment = require('moment'); //Node环境特有的require语句

var formatDate = function(date) {
    return moment(date).format('MMMM Do YYYY, h:mm:ss a');
};

module.exports = formatDate
```

我们还有一个简单的 main.js 文件，该文件会调用 `formatDate(..)` 方法，以格式化当前时间：

```
var formatDate = require('./dateFormatter.js');
console.log(formatDate(Date.now()));
```

当在服务器端运行 main.js 时（使用 Node.js），我们会得到如下输出：

```
$ node main.js
July 25th 2015, 11:27:27 pm
```

Browserify（http://browserify.org/）是一个编译 CommonJS 模块的工具，该工具可以将所有引入的 Node 模块打包起来，在浏览器中运行。借助 Browserify，我们可以输出一个对浏览器友好的 JavaScript 文件：

```
$ browserify main.js > bundle.js
```

当在浏览器中打开 bundle.js 文件时，我们可以在浏览器控制台中看到相同的日期信息（如图 3-2 所示）。

```
<script src="bundle.js"></script>
```



图 3-2：运行 bundle.js 时的浏览器控制台输出

在继续介绍之前，我们先暂停一下，回想一下刚才发生的事情。尽管这只是一个简单的例子，但有着深远的影响。有了简单的构建工具后，我们就可以轻松实现从服务器端到客户端的逻辑共享。这就带来了许多可能性，我们将在本书的第二部分深入探讨。

3.2 为每个特定环境提供shim

客户端和服务器端的 JavaScript 环境存在许多区别。在客户端，我们拥有全局对象（如 `window`）以及各种 API，其中包括 localStorage、History API 以及 WebGL。而在服务器端，我们在一个请求 / 响应生命周期的上下文环境中工作，而且服务器端还拥有自身的全局对象。

如果在浏览器中运行以下代码，那么将会返回浏览器当前的 URL 地址。改变这个属性的值将会导致页面重定向：

```
console.log(window.location.href);
window.location.href = 'http://www.oreilly.com'
```

在服务器端运行同样的代码则会返回一个错误：

```
> console.log(window.location.href);
ReferenceError: window is not defined
```

这是因为 `window` 在服务器端不是一个全局对象。为了在服务器端实现相同的重定向功能，我们必须在响应对象中写入头部信息，包括一个指明 URL 重定向的状态码（如 302）以及客户端将要跳转的地址 `location`：

```
var http = require('http');
http.createServer(function (req, res) {
   console.log(req.path);
   res.writeHead(302, {'Location': 'http://www.oreilly.com'});
   res.end();
}).listen(1337, '127.0.0.1');
```

正如我们看见的那样，服务器端的代码看起来和客户端的差异很大。那么如何让同一份代码在两端都能运行呢？

我们有两种可选方案。第一种方案是将重定向的逻辑分离到一个独立的模块中，这个模块

需要知道当前的运行环境。应用的剩余代码只需要调用该模块即可，从而实现具体环境的完全隔离：

```
var redirect = require('shared-redirect');

// 执行一些有趣的应用逻辑,判断是否需要进行重定向

if(isRedirectRequired){
  redirect('http://www.oreilly.com');
}

// 继续执行有趣的应用逻辑
```

这种方式使得应用逻辑变得与环境无关，可以同时在客户端和服务器端运行。虽然 `redirect(..)` 函数的实现需要考虑到特定环境，但其逻辑是独立的，不会影响到应用的其他方面。以下是 `redirect(..)` 函数的一种实现方式：

```
if (typeof window !== 'undefined') {
  window.location.href = 'http://www.oreilly.com'
}else{
  this._res.writeHead(302, {'Location': 'http://www.oreilly.com'});
}
```

注意，这个函数必须判断 `window` 对象是否存在，并根据情况判断是否使用它。

另一种方法是，在客户端使用服务器端的响应对象接口，但需要进行 shim，实质上还是调用了 `window` 属性。通过这种方式，应用代码只需要调用 `res.writeHead(..)` 即可，但是这在浏览器中会转为调用 `window.location.href` 属性。我们将在本书的第二部分中更详细地分析这种实现方式。

3.3 小结

在本章中，我们探讨了两种不同的同构 JavaScript 代码。我们研究了如何简单地使用 Browserify 这样的工具，将与环境无关的 Node 模块代码转换到浏览器中。此外，还探讨了与环境相关的代码是如何为特定环境实现 shim 的，以允许代码在客户端和服务器端重用。现在是时候进行更深层次的讨论了。下一章将超越服务器端渲染，研究如何在不同的解决方案中使用同构 JavaScript。我们将探索创新的、前瞻性的应用架构，这些架构可以使用 JavaScript 来完成一些新奇的事情。

第 4 章

超越服务器端的渲染

Maxime Najim

应用具有不同的规模。在前面的介绍性章节中，我们主要关注的是也可以在服务器端进行渲染的 SPA。服务器渲染首屏页面的目的是改善页面的感知加载时间并实现搜索引擎优化。这些讨论的焦点是传统的应用架构，即客户端发起一个 REST 调用，然后被路由到某个无状态的后端服务器中，接着服务器依次查询数据库，并将结果返回到客户端（如图 4-1 所示）。

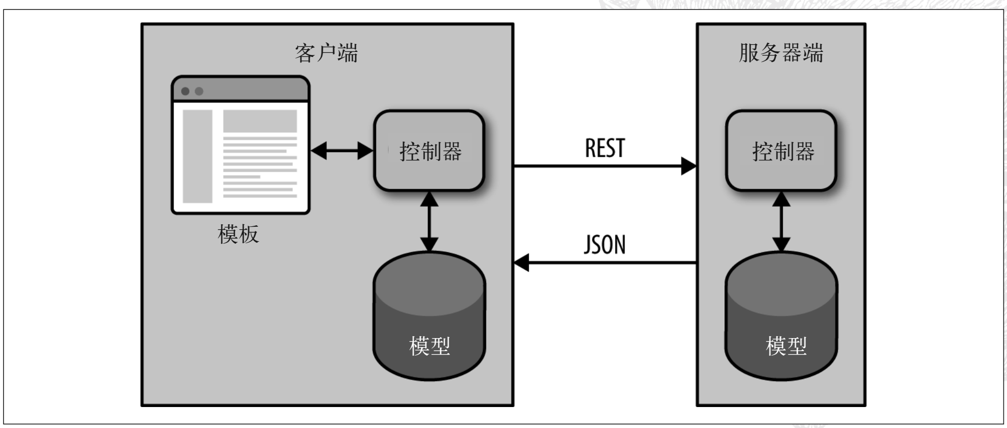


图 4-1：传统的 Web 应用架构

这种方式非常适合传统的电商 Web 应用。然而，还存在一种通常被称为“实时应用”的应用。事实上，我们可以认为有两种同构 JavaScript 应用：一种是可以在服务器端渲染的 SPA，另一种则是将同构 JavaScript 用在实时、离线及数据同步等功能中的应用。

Matt Debergalis 将实时应用描述为应用架构的丰富历史中的一个自然演化步骤。他认为应用架构的变化是由以下因素驱动的：廉价 CPU 的新时代、互联网，以及移动网络的出现。以上这些变化都促进了新应用架构的发展。然而，尽管我们已经看到了更复杂的、支持实时更新和协同工作的实时应用，但目前的大多数应用依然是支持服务器端渲染的 SPA。但是，我们依然认为实时应用对未来的应用架构至关重要，而且也非常切合我们对同构 JavaScript 应用的讨论。

4.1 实时Web应用

实时应用拥有丰富的界面交互和协作元素，允许用户之间共享数据。聊天应用 Slack、文档共享应用 Google Docs，以及提供共享车服务并实时向所有用户显示当前位置附近可用司机的 Uber，这些都属于实时 Web 应用。对于这类应用，我们最终设计并实现了将数据从服务器端推送到客户端的一种方式，以便显示其他用户的变化。我们还需要实现一种更新方式，以便当数据从服务器端发送过来时，客户端可以反应性地更新每位用户的屏幕内容。大部分的实时应用有着类似的功能。这些应用必须具备以下功能：用于监听数据库变化的机制；在推送技术之上运行的某种协议（如 Websocket），以便将数据推送到客户端（或者使用长轮询来模拟服务器端的数据推送，即服务器端需要一直保持请求打开的状态，直到新的数据准备好并发送到客户端）；以及某种客户端缓存技术，以避免在重绘屏幕时频繁地在服务器端往返数据。

在图 4-2 中，我们可以看到用户与视图的交互是如何影响数据流动的。我们还可以看到，来自其他客户端的变化是如何传播到所有用户的，以及在接收到服务器端发送的数据变化后，视图是如何重新进行渲染的。这种架构催生了三种有趣的同构概念：同构 API、双向数据同步，以及在服务器端进行的客户端仿真。

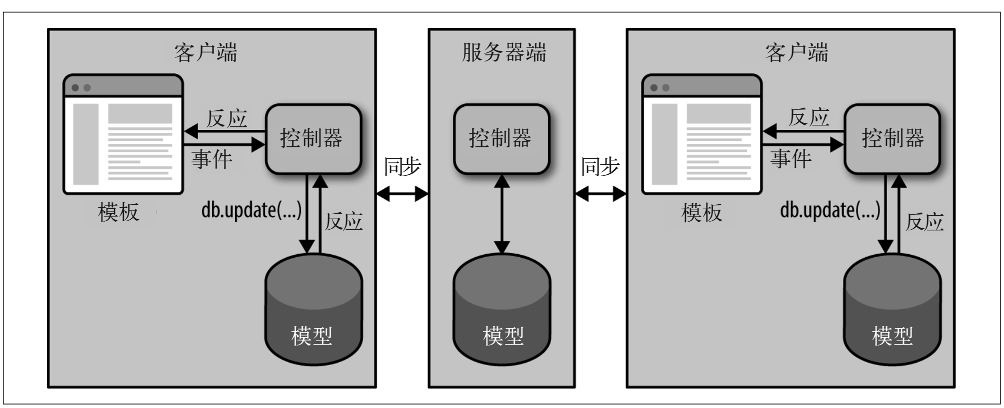


图 4-2：实时 Web 应用架构

4.1.1 同构API

在同构的实时应用中，客户端和本地数据缓存的交互方式与服务器端和后台数据库的交互方式十分类似。服务器端代码可以对数据库执行一条语句，客户端代码可以使用相同的数据库 API 来执行相同的语句，从而向内存中的缓存请求数据。客户端 API 与服务器端 API 之间的这种对称性通常被称为**同构 API**。同构 API 可以将开发人员从同时兼顾不同的数据访问策略中解放出来。更重要的是，同构 API 可以让客户端和服务器端同时运行应用的核心业务逻辑（尤其是在模型层）与渲染逻辑。用同构 API 访问数据可以让我们在服务器端和客户端之间共用一套代码来验证数据更新、访问并存储数据，以及改变数据。过去，我们需要编写多个版本的代码来实现同样的功能，并以不同的方式来测试，而且在改变数据模型时必须修改两遍代码；而现在，我们只需要使用一致的 API 即可免除这些烦恼。同构 API 遵循了 DRY 原则，这种实现方式是智能化的。

4.1.2 双向数据同步

实时应用的另一个重要方面是服务器的数据库与客户端的本地缓存之间的同步。从客户端到服务器的更新应该被同步到客户端的本地缓存中，反之亦然。实时同构框架中一个很好的例子是 Meteor.js，该框架允许开发者编写可以同时在服务器端和客户端运行的代码。Meteor 遵循“处处是数据库”（Database Everywhere）的原则。在 Meteor 中，客户端和服务器端可以使用同一套同构 API 访问数据库。Meteor 还在客户端使用了数据库抽象，比如 minimongo，并在 DDP（dynamic data protocol，动态数据协议）之上使用了 observable collection，目的是保持数据在服务器端和客户端的同步。驱动 UI 变化的正是客户端数据库（而非服务器端数据库）。客户端数据库需要进行数据的“懒”同步，以保证服务器端的数据可以及时更新。这使得客户端可以离线工作，并在本地环境中继续处理用户数据的变化。在收到服务器端的确认返回之前，客户端的每一次写入操作都可以选择性地推测是写入客户端的本地缓存中的。Meteor 有一种内建的延迟补偿机制，如果向服务器写入数据库失败或者写入被另一个客户端覆盖，那么就会刷新推测缓存。

4.1.3 在服务器端进行客户端仿真

在极端情况下，同构 JavaScript 应用可能需要为每一个客户端会话运行独立的进程。这使得服务器端可以查看应用加载的数据并主动向客户端发送数据，本质上就是在服务器端模拟 UI。这种技术已经被运用到了 Asana（https://asana.com/）应用中。Asana 是一个协同项目管理工具，是基于一个被称为 Luna 的内部闭源框架所构建的。Luna 是为编写实时 Web 应用量身定做的，用法与 Meteor.js 类似。它提供了一套通用的同构 API，用于在客户端和服务器端访问数据。然而，Luna 真正的特别之处在于，它可以在服务器端运行应用的一份完整的副本。在服务器端，Luna 通过客户端的方式执行同一份 JavaScript 代码，从而模拟客户端的运行。当用户在 Asana 的界面中点击某个地方时，客户端的 JavaScript 事件会同

步到服务器端。通过执行所有的视图与事件，服务器端维护着一份用户状态的完整版本，但其输出仅仅是简单的 HTML 内容。

然而，在 Asana 最近更新的一篇技术博文（https://blog.asana.com/2015/05/the-evolution-of-asanas-luna-framework/）中，他们表示正在逐步停用这种客户端 / 服务器端仿真的方式。这是因为其性能存在问题，尤其是在服务器需要模拟 UI 的多种状态，以便客户端可以立刻预测和预加载数据的情况下。这篇文章还提到了移动客户端的版本问题，如果客户端运行的代码是旧版本，那么就会导致仿真变得复杂，因为服务器端实际运行的代码版本可能会和客户端的代码不一致。

4.2 小结

同构 JavaScript 是在两端共享应用代码的一种尝试。通过了解实时的同构框架 ，我们找到了共享应用逻辑的不同方案。这些框架采取的方法更加前沿，而不仅仅是在服务器端渲染一个 SPA 那么简单。关于这些概念的讨论已经很多了，我们希望这些讨论能为同构 JavaScript 的方方面面提供一个全面的介绍。在本书的下一部分中，我们将基于这些关键概念来创建我们的第一个同构应用。

第二部分

构建第一个应用

优秀的软件设计的关键在于：知道需要抽象什么、何时进行抽象，以及在何处进行抽象。如果抽象过度或者过早地进行抽象，那么结果就是添加了一层没多少价值的复杂度。如果抽象程度不够或者没有在恰当的地方进行抽象，那么就会造成应用结构脆弱，难以支撑规模扩张。当你可以在两者之间取得完美的平衡时，软件才会产生真正的美。这就是软件设计的艺术，工程师欣赏这种艺术，正如艺术评论家欣赏毕加索或者伦勃朗那样。

在本书的这一部分中，我们将尝试创造美。许多前人已经走过了我们这条路，有人成功，也有人失败。而我有幸经历过这两种情况，虽然其中的失败比成功多，但我在每一次失败中都有所收获。带着从这些经历中得到的知识，我将带领大家穿过设计过程中的层层障碍，实现一个轻量级的同构 JavaScript 应用框架。

这一过程并不轻松，大部分人都会在形式、结构和抽象上犯错，但我已经准备好迎接这个挑战了，希望你也如此。虽然最后或许不能取得完美的结果，但我们可以吸取教训并应用在未来的同构 JavaScript 实践中，并且我们可以将基础打好，方便以后扩展。毕竟这仅仅是软件进化过程中的一小步，也是我们学习过程中的一小步。你准备好了吗？接下来，我们将从头开始逐步向前迈进。

第 5 章

起步

Jason Strimpel

我们已经对同构 JavaScript 有了充分的了解，现在是时候从理论转为实践了。在本章中，我们将奠定基础，并在此之上逐步建立一个具有完整功能的同构 JavaScript 应用。这个基础主要由下列技术构成。

- Node.js（https://nodejs.org/）将作为应用的服务器端运行环境。它是一个基于 Chrome 的 JavaScript 运行环境构建的平台，可以让你轻松地构建快速的、可扩展的网络应用。Node.js 使用了事件驱动的、非阻塞的 I/O 模型，具备轻量级和高效的特点，非常适合构建在分布式设备上运行的数据密集型的实时应用。
- Hapi.js（https://hapijs.com/）将用于驱动应用中的 HTTP 服务器部分。它是一个功能丰富的框架，可以用于构建应用与服务。Hapi.js 帮助开发者集中精力编写可重用的应用逻辑，而不是将时间花费在基础设施的构建上。
- Gulp.js（http://gulpjs.com/）将用于编译我们的 JavaScript 代码（将 ES6 转译为 ES5），针对浏览器环境进行打包，并管理我们的开发工作流程。它是一个基于 Node 中 stream 模块的流处理构建系统：所有的文件操作都在内存中完成，除非你命令其进行文件写入，否则它不会修改任何文件内容。
- Babel.js（https://babeljs.io/）让我们可以利用 ES6 的特性和语法来编写代码，然后帮我们将代码编译为兼容 ES5 的可发布版本。它是一个用于编写新一代 JavaScript 代码的编译器。

ES6 和 ES2015

ES6 又称为 ES2015。在制定规范和审核流程的后期，ES6 更名为 ES2015，但这个版本一般被俗称为 ES6。如果想了解更多关于 JavaScript 版本命名约定的内容，请参见“ECMAScript 6 support in Mozilla”（https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla）。

直接安装整个项目

如果你对于 Node、`npm` 和 Gulp 的使用已经有所了解，那么你可以跳过本章，直接通过运行 `npm install thaumoctopus-mimicus@"0.1.x"` 命令来安装我们最终生成的项目代码。

5.1　Node的安装和运行

Node 的安装非常简单。Node 可以在 Linux、Mac OS、Windows 和 Unix 平台上运行。你可以选择在终端编译源码来安装，使用包管理器（如 `yum`、`homebrew` 和 `apt-get` 等）安装，或者在 Mac OS、Windows 平台上直接通过安装程序进行安装。

5.1.1　从源码安装

本节概述了如何通过编译源码来安装 Node。尽管我强烈推荐通过安装程序或包管理器的方式进行安装，但如果你是热衷于通过源码安装软件的少数派，那么这一节的内容可以帮到你。否则，请跳到下一节继续阅读。

首先，从 Nodejs.org 下载源代码：

```
$ wget http://nodejs.org/dist/v0.12.15/node-v0.12.15.tar.gz
```

Node 版本

在编写本书时，node-v0.12.15.tar.gz 是最新的稳定版本。你可以通过 https://nodejs.org/download/release/ 来检查最近更新的版本，并将 `wget` 示例命令中的版本号 `v0.12.15` 替换为最新的稳定版本。

接下来，你需要从下载的文件中解压出源代码：

```
$ tar zxvf node-v0.12.15.tar.gz
```

上述命令解压并提取出 Node 源代码。要想详细了解有关 `tar` 命令的更多选项，可以在终端中执行 `man tar` 来查看。

现在电脑上已经有源代码了，你需要运行源码目录中的配置脚本。这个脚本会在你的系统中寻找相应的 Node 依赖关系，并在缺少依赖时通知你。

```
$ cd node-v0.12.15
$ ./configure
```

一旦找到所有的依赖关系，就可以将源码编译为二进制文件了。这项工作可以通过 `make` 命令来完成：

```
$ make
```

最后一步是运行 `make install` 命令。由于 Node 的安装是全局性的，因此需要 `sudo` 命令授予管理特权：

```
$ sudo make install
```

如果一切顺利，当检查 Node.js 版本时，你应该能看到如下的输出信息：

```
$ node -v
v0.12.15
```

5.1.2　与Node REPL交互

Node 是一个带有 REPL（read-eval-print loop，读入－求值－打印的循环）的运行时环境，REPL 就是一个可以编写 JavaScript 代码的 JavaScript shell，在你按下回车键后，它会对语句进行求值。这就像是在你的终端中拥有 Chrome 开发者工具中的控制台一样。你可以尝试输入几条基本的命令来试验一下：

```
$ node
> (new Date()).getTime();
1436887590047
>
(^C again to quit)
>
$
```

这个功能可以帮助我们测试和调试代码片段。

5.1.3　使用npm管理项目

`npm`（http://www.npmjs.com/）是 Node 的包管理工具。`npm` 让开发者可以在不同的项目中轻松地重用代码，并与其他的开发者共享代码。`npm` 是集成在 Node 安装包中的，因此，在安装 Node 的同时，你其实已经装好了 `npm`：

```
$ npm -v
2.7.0
```

网上有大量关于 npm 的教程和丰富的文档（https://docs.npmjs.com/）可供参考，这已经超出了本书的讨论范围。在本书的示例中，我们主要会用到 package.json 文件，该文件中包含了项目的元数据以及 init 和 install 命令。我们将通过项目中的实际例子来学习如何利用 npm。

5.2 建立应用项目

当谈及软件项目管理时，除了源代码管理之外，另一个最重要的方面就是打包、共享和部署代码的途径。在本节中，我们将使用 npm 来建立项目。

5.2.1 初始化项目

npm 的 CLI（command-line interface，命令行界面）是一个终端程序，可以让你快速地执行管理项目和软件包的命令。其中一个命令是 init。

> 如果你已经熟悉 npm init 命令的使用方法或者想直接查看源代码，那么可以直接跳到下一节。

init 是一个交互式的命令，该命令会向你询问一系列问题，并为项目创建一个 package.json 文件。该文件中包含了项目的元数据（包括包名称、版本号、依赖关系等）。这些元数据用于将软件包发布到 npm 仓库中，以及从仓库安装软件包。让我们实际运行一次该命令：

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
```

> **项目目录**
>
> 出于简洁的目的，上述的终端代码示例省略了计算机名、路径和用户名（如 fantasticplanet:thaumoctopus-mimicus jstrimpel $）。接下来的所有终端命令都是从项目目录中执行的。

你将看到的第一个提示是让你输入包名称。默认值是当前目录的名称，在我们的示例中是 thaumoctopus-mimicus。

```
name: (thaumoctopus-mimicus)
```

项目名称

整个第二部分构建的项目的名称就是 thaumoctopus-mimicus。我们会为这一部分的每一章订立一个次要版本，例如本章的版本号为 0.1.x。

点击回车键继续。下一个提示是输入版本号：

```
version: (0.0.0)
```

目前可以采用 `0.0.0` 作为版本号，因为才刚刚开始，还没有任何有价值的内容可以发布。接下来的提示是输入应用的描述：

```
description:
```

可以输入 Isomorphic JavaScript application example。下一步需要输入项目的入口点：

```
entry point: (index.js)
```

当其他用户在自己的源代码中引入你的包时，入口点就是对应的文件。目前使用 `index.js` 就可以了。下一个提示是输入 `test` 命令。留空即可（我们不会具体探讨测试的内容，因为这超出了本书的讨论范围）：

```
test command:
```

接下来将会提示你输入项目的 GitHub 仓库地址。这里默认提供的是 https://github.com/isomorphic-javascript-book/thaumoctopus-mimicus.git，也就是本项目的仓库地址。你的显示很可能是空白。

```
git repository:
  (https://github.com/isomorphic-javascript-book/thaumoctopus-mimicus.git)
```

下一个提示是输入项目的关键词：

```
keywords:
```

输入 isomorphic javascript。然后，你会被问及作者的名字：

```
author:
```

在这里输入你的名字。最后一个提示是关于开源协议的。默认值可能是 `(ISC) MIT` 或者 `(ISC)`，具体会根据 NPM 的版本而定，这刚好是我们所需要的：

```
license: (ISC) MIT
```

如果现在切换到项目目录中，你会看到一个 package.json 文件，文件的内容如例 5-1 所示。

例 5-1 通过 `npm init` 创建的 package.json 文件

```
{
  "name": "thaumoctopus-mimicus",
  "version": "0.0.0",
  "description": "Isomorphic JavaScript application example",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/isomorphic-javascript-book/thaumoctopus-mimicus.git"
  },
  "keywords": [
    "isomorphic",
    "javascript"
  ],
  "author": "Jason Strimpel",
  "license": "MIT",
  "bugs": {
    "url":
    "https://github.com/isomorphic-javascript-book/thaumoctopus-mimicus/issues"
  },
  "homepage": "https://github.com/isomorphic-javascript-book/thaumoctopus-mimicus"
}
```

5.2.2 安装应用服务器

在上一节中，我们初始化了项目，并创建了一个包含项目元数据的 package.json 文件。虽然这是一个必经的过程，但并没有为我们的项目添加任何功能，这个过程本身也不是很有趣。因此，我们继续下一步，来实现一些更有意思的事情。

所有的 Web 应用都需要某种应用服务器，同构 JavaScript 应用也不例外。无论是仅仅提供静态文件，还是基于服务请求与业务逻辑组装 HTML 文档响应，应用服务器都是一个必不可少的部分，因此我们选择从这里开始编码旅程。我们将会使用 hapi（http://hapijs.com/）来创建应用服务器。安装 hapi 是一个非常轻松的过程：

```
$ npm install hapi --save
```

上述命令不仅安装了 hapi，还在项目的 package.json 文件中添加了一条依赖项。这样做是为了方便他人（当然也包括你自己）在安装你的项目时，可以同时安装运行该项目所需要的所有依赖。

既然已经安装了 hapi，接下来就可以编写第一个应用服务器了。第一个示例的目标是响应

hello world 文本。在你的 index.js 文件中输入例 5-2 中的代码。

例 5-2　hapi 输出 hello world 的示例

```
var Hapi = require('hapi');

// 创建一个服务器,并配置主机名与端口
var server = new Hapi.Server();
server.connection({
    host: 'localhost',
    port: 8000
});

// 添加路由
server.route({
    method: 'GET',
    path:'/hello',
    handler: function (request, reply) {
       reply('hello world');
    }
});

// 启动服务器
server.start();
```

在终端中输入 `node .` 来启动该应用，并在浏览器中打开 http://localhost:8000/hello。如果你能看到 hello world 的字样，那么恭喜你，你成功了！否则，请回到上一节中检查你或者我是否遗漏了某些步骤。如果所有的方法都失败了，你可以尝试将这段存放在 gist 上的代码（https://gist.github.com/jstrimpel/3b2770558b2b397f616f）直接复制到你的 index.js 文件中。

5.2.3　编写下一代的JavaScript（ES6）

ECMAScript 6（http://www.ecmascript.org/，简称 ES6）是编写本书时 JavaScript 的最新版本，该版本为这门语言添加了许多新功能。这一规范是在 2015 年 6 月 17 日被批准并公布的。虽然大家对某些新功能的意见不一（比如类的引入），但总体来说，ES6 还是深受欢迎的。在编写本书时，许多公司已经广泛采用 ES6。

和许多非原生实现一样，ES6 中的类仅仅是原型继承的语法糖。之所以将这项特性添加到 JavaScript 中，很可能是为了让语言能够提供一个共同的参照框架，以吸引那些来自传统类继承语言的开发者。类的使用将贯穿本书。

业内已经达成了普遍共识，开始利用 ES6 提供的新特性，虽然这些功能尚未得到广泛支持，但现在可以通过将 ES6 编译为支持度较高的 ECMAScript 版本（即 ES5）来解决这个问题。编译过程的一部分内容就是为目标版本中缺失的 ES6 特性提供 polyfill。鉴于这种行业趋势以及现成的编译支持，我们将在本书后续的示例中利用 ES6 进行编写。现在我们先将上一个示例中的代码修改为 ES6 版本。

首先，我们要改变在 index.js 文件中导入 hapi 依赖的方式。将 index.js 文件中的第一行替换为如下内容：

```
import Hapi from 'hapi';
```

ES6 为 JavaScript 引入了模块系统的概念，这个概念从 JavaScript 诞生起就一直缺失。在原生模块接口一直缺失的情况下，其他一些模式的出现填补了这一空白，比如 AMD 和 CommonJS。在例 5-2 原版的 index.js 文件中，`require` 语法就遵循了 CommonJS 规范。

> 之所以没有原生模块接口，是因为在创造 JavaScript 时，人们并没有料到它能够像今天这样驱动应用。设计 JavaScript 的目的仅仅是为了提高网页的吸引力，并丰富客户端体验。

接下来需要修改 `server` 变量声明。将文件的第二行替换为：

```
const server = new Hapi.Server();
```

在 ES6 中，`const` 是声明变量的一种新方法。当一个变量使用 `const` 声明时，不能修改其引用。举例来说，如果通过 `const` 创建一个对象，当你想要向对象中添加属性时，JavaScript 不会抛出异常，因为引用没有被修改。然而，如果你试图将变量指向另一个对象，这将会导致错误，因为引用被修改了。我们在这里使用 `const` 是因为不希望出现不小心将 `server` 变量指向另一个引用的情况。

> `const` 不是指变量值不能被修改，而是指在内存中存放了一个值的常引用。

例 5-3 展示了修改完成后的 index.js 文件。

例 5-3　ES6 版本的 hapi 输出 hello world 的示例

```
import Hapi from 'hapi';

// 创建一个服务器,并配置主机名与端口
const server = new Hapi.Server();
server.connection({
    host: 'localhost',
    port: 8000
});

// 添加路由
server.route({
    method: 'GET',
    path:'/hello',
    handler: function (request, reply) {
```

```
        reply('hello world');
    }
});

// 启动服务器
server.start();
```

我们已经将 index.js 文件修改为使用最新、最好的 ES6 语法，现在是时候尝试运行了。运行的 Node 版本不同，运行的结果也会有所区别，这也就是说，浏览器可能不会渲染 hello world 了。这是因为你运行的 Node 版本不一定支持 ES6。好在这个问题是可以解决的，因为编译器可以帮助我们将 ES6 代码编译为可以在 Node 4+ 中运行的代码。

5.2.4　将ES6编译为ES5

在本章的开头部分，我们提到了一个名为 Babel（https://babeljs.io/）的 JavaScript 编译工具。我们将在本书的余下内容中使用 Babel 来编译 JavaScript，但在此之前，我们需要先安装 Babel 和 ES2015 的转换包。

```
$ npm install --save-dev babel-cli babel-preset-es2015
```

> **Babel 的插件与预设**
>
> 默认情况下，Babel 自身不会进行任何转换，它使用插件来转换代码。在上述命令中，我们安装了一个**预设**，这个预设对应一个或者一系列配置好的插件，用于将 ES6 代码转换为 ES5 代码，以便让代码可以在旧版本的浏览器或者旧版本的 Node 环境下运行。

就像 hapi 的安装命令一样，上述命令会将 Babel 作为依赖项添加到你的 package.json 文件中。唯一不同的地方是，我们传递的参数是 `--save-dev` 而不是 `--save`，因此这一项依赖会被归到 package.json 的 `devDependencies` 属性当中，而不是在 `dependencies` 当中。这样做的原因是，运行你的应用是不需要 Babel 的，但它是开发过程中所需要的依赖。这种区分可以让使用者清楚地知道生产环境和开发环境所需要的依赖分别是什么。当你的项目通过 `--production` 标记安装时，Babel CLI 将不会被安装。

现在我们已经成功地安装了 Babel，接下来就可以编译 index.js 了。在此之前，我们应该先对项目的目录结构进行一些小调整。

如果按照现有的项目结构来编译 index.js 文件，那么我们的源代码将会被覆盖——在没有备份的情况下丢失源代码可不是什么好事。解决该问题的方法是，给 Babel 指定一个输出文件。通常的约定是将项目发布的目录称为 dist，源码目录称为 src。因为我不是一个喜欢发明新标准的人，所以我们还是遵循这种约定：

```
$ mkdir dist
$ mkdir src
$ mv index.js src/index.js
```

将项目的目录结构调整为适合编译的结构后，我们需要修改 package.json 文件。首先，通过将 "main": "index.js," 这一行修改为 "main":"./dist/index.js",，我们将 main 属性指向新的发布目录，这样做是为了告诉 Node 应用的新入口点，以便在执行 node . 命令时能够加载正确的脚本。现在我们已经准备好进行第一次编译了！在命令行中输入如下命令：

```
$ babel src/index.js --out-file dist/index.js
```

Command Not Found!

如果在执行命令时出现“Command not found!”错误，那么你可能需要指明 Babel 可执行程序的路径，即 ./node_modules/.bin/babel，或者通过 npm install -g babel-cli 来全局性地安装 babel 命令。

上述命令取得了源代码 src/index.js，然后进行编译，并创建了发行版的目标文件 dist/index.js。如果一切顺利，现在应该可以再次使用 node . 命令来启动我们的服务器了。

Babel 和 ES6 特性

在上面这几节中，我们仅仅是浅显地介绍了 Babel 和 ES6。要想了解更多信息，请访问网址 https://babeljs.io/。

5.2.5　建立开发流程

现在我们已经有能力随时编译代码，并通过命令行来重启服务器，这是一项了不起的成就。但在开发过程中，每次都要停下来执行这两条命令会让你很快就疲惫不堪。好在我们不是第一个需要自动化重复任务并优化工作流程的人，现在已经有好几种自动化方案可供选择了。我们当然想要选择最新、最强大的方案，以便我们的代码在未来半年内不至于变得落后。最近新增到 JavaScript 构建方案的工具是 Gulp（http://gulpjs.com/），这是一个流处理构建系统，你可以在 Gulp 中创建任务，并使用其社区驱动的生态系统提供的各种插件。这些任务可以链接到一起，从而创建出你的构建流程。听起来很棒对吧？事不宜迟，首先需要安装 Gulp：

```
$ npm install gulp --save-dev
```

接下来需要在项目的根目录中创建一个 gulpfile.js 文件，并定义一项默认任务，如例 5-4 所示。

例 5-4　在 gulpfile.js 文件中创建默认的 Gulp 任务

```
var gulp = require('gulp');

gulp.task('default', function () {
  console.log('default task success!');
});
```

安装好 Gulp 并定义了一项默认任务后，现在来运行 Gulp：

```
$ gulp
[20:00:11] Using gulpfile ./gulpfile.js
[20:00:11] Starting 'default'...
default task success!
[20:00:11] Finished 'default' after 157 μs
```

为了方便代码块的排版，Gulp 输出中的所有绝对路径都会被转换为相对路径（如 ./gulpfile.js）。但在你的屏幕显示的输出内容中，你看到的将会是绝对路径。

Command Not Found!

如果执行 `gulp` 命令时出现了“Command not found!”错误，那么你可能需要指明 Gulp 可执行程序的路径，即 ./node_modules/.bin/gulp，或者通过 `npm install -g gulp` 来全局性地安装 `gulp` 命令。

成功了！ Gulp 已经可以运行了。现在我们来创建一项实质性的任务，比如编译源代码。这需要借助 `gulp-babel` 插件（https://www.npmjs.com/package/gulp-babel）来实现，我们可以通过如下方式进行安装：

```
$ npm install gulp-babel --save-dev
```

现在需要在 gulpfile.js 文件中修改默认任务来处理编译。用例 5-5 所示的代码替换你之前创建的文件内容：

例 5-5　使用 `gulp-babel` 编译源代码

```
var gulp = require('gulp');
var babel = require('gulp-babel');

gulp.task('compile', function () {
  return gulp.src('src/**/*.js')
    .pipe(babel({
       presets: ['es2015']
    }))
    .pipe(gulp.dest('dist'));
});

gulp.task('default', ['compile']);
```

现在再次运行 Gulp：

```
$ gulp
[23:55:13] Using gulpfile ./gulpfile.js
[23:55:13] Starting 'compile'...
```

```
[23:55:13] Finished 'compile' after 251 ms
[23:55:13] Starting 'default'...
[23:55:13] Finished 'default' after 17 μs
$
```

这种做法很不错，但还没有解决我们之前的问题。我们真正做到的事情只是减少了在终端需要输入的代码量。为了让 Gulp 的引入发挥真正的价值，我们需要自动化编译流程，并在每次保存源代码文件时自动重启服务器。

1. 监听源代码变化

Gulp 包含一个内建的文件监听方法 `gulp.watch`（https://github.com/gulpjs/gulp/blob/master/docs/API.md#gulpwatchglob%E2%80%94opts-tasks-or-gulpwatchglob%E2%80%94opts-cb），该方法接收的参数包括文件匹配模式（glob）、可选参数，以及一个任务列表或回调函数。当发生变化的文件与 glob 匹配时，就会执行任务列表或回调函数。我们正需要通过这种方式来运行 Babel 任务，所以现在可以进行配置了。将下列任务添加到 gulpfile.js 文件中：

```
gulp.task('watch', function () {
  gulp.watch('src/**/*.js', ['compile']);
});
```

这样应该就能完成我们一直以来的目标了，所以将任务添加到默认任务中：

```
gulp.task('default', ['watch', 'compile']);
```

如果现在在终端运行 `gulp`，然后修改代码文件，那么你应该可以看到类似以下这样的输出结果：

```
$ gulp
[00:04:35] Using gulpfile ./gulpfile.js
[00:04:35] Starting 'watch'...
[00:04:35] Finished 'watch' after 12 ms
[00:04:35] Starting 'compile'...
[00:04:35] Finished 'compile' after 114 ms
[00:04:35] Starting 'default'...
[00:04:35] Finished 'default' after 16 μs
[00:04:39] Starting 'compile'...
[00:04:39] Finished 'compile' after 75 ms
```

这样就方便多了，因为不再需要在每次修改源代码时都运行一次编译命令。现在只需要让服务器自动重启即可。

2. 在发行版内容变化时重启服务器

为了监听发行版的文件 dist/index.js，我们需要借助 `gulp-nodemon`（https://www.npmjs.com/package/gulp-nodemon），这是一个 nodemon（http://nodemon.io/）的包装层。nodemon 是一款实用工具，用于监听源代码的变化，并在发生变化时自动重启服务器。在使用 `gulp-nodemon` 之前，先要进行安装：

```
$ npm install gulp-nodemon --save-dev
```

接下来需要安装 `run-sequence`：

```
$ npm install run-sequence --save-dev
```

这个包以特定的顺序执行一系列的 Gulp 任务。我们必须借助这个包，因为我们需要确保在服务器启动之前已经生成发行版的文件。

现在要对 gulpfile.js 进行一些必要的修改，告诉 `gulp-nodemon` 何时应该重启服务器，并引入 `run-sequence` 包。在文件中添加以下内容：

```
var nodemon = require('gulp-nodemon');
var sequence = require('run-sequence');

gulp.task('start', function () {
  nodemon({
    watch: 'dist',
    script: 'dist/index.js',
    ext: 'js',
    env: { 'NODE_ENV': 'development' }
  });
});
```

最后，将新的 nodemon 监听任务添加到默认任务中，并使用 `run-sequence` 指定任务的执行顺序：

```
gulp.task('default', function (callback) {
  sequence(['compile', 'watch'], 'start', callback);
});
```

现在，当运行默认任务并修改源代码时，你应该能看到类似以下这样的输出结果：

```
$ gulp
[16:51:43] Using gulpfile ./gulpfile.js
[16:51:43] Starting 'default'...
[16:51:43] Starting 'compile'...
[16:51:43] Starting 'watch'...
[16:51:43] Finished 'watch' after 5.04 ms
[16:51:44] Finished 'compile' after 57 ms
[16:51:44] Starting 'start'...
[16:51:44] Finished 'start' after 849 μs
[16:51:44] Finished 'default' after 59 ms
[16:51:44] [nodemon] v1.4.0
[16:51:44] [nodemon] to restart at any time, enter `rs`
[16:51:44] [nodemon] watching: ./dist/**/*
[16:51:44] [nodemon] starting `node dist/index.js`
[16:51:47] Starting 'compile'...
[16:51:47] Finished 'compile' after 19 ms
[16:51:48] [nodemon] restarting due to changes...
[16:51:48] [nodemon] starting `node dist/index.js`
```

gulpfile.js 文件的完整版如例 5-6 所示。

例 5-6　gulpfile.js 文件的完整版

```
var gulp = require('gulp');
var babel = require('gulp-babel');
var nodemon = require('gulp-nodemon');
var sequence = require('run-sequence');

gulp.task('compile', function () {
  return gulp.src('src/**/*.js')
    .pipe(babel({
      presets: ['es2015']
    }))
    .pipe(gulp.dest('dist'));
});

gulp.task('watch', function () {
  gulp.watch('src/**/*.js', ['compile']);
});

gulp.task('start', function () {
  nodemon({
    watch: 'dist',
    script: 'dist/index.js',
    ext: 'js',
    env: { 'NODE_ENV': 'development' }
  });
});

gulp.task('default', function (callback) {
  sequence(['compile', 'watch'], 'start', callback);
});
```

5.3　小结

本章介绍了许多内容，从 Node 的安装到为第一个 hapi 应用服务器优化开发流程，其中还包括了 ES6 的使用。以上这些都是为现代 JavaScript 应用开发做的准备。同时，这些知识也为我们将在第 6 章中构建的示例应用建立了基础。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.1.x"` 命令，可以安装本章的完整代码示例。

第 6 章

提供第一份HTML文档

Jason Strimpel

在创建同构 JavaScript 框架或应用时，大部分人都会选择从客户端逻辑开始编写，随后尝试将这套方案整合到服务器端。这可能是因为他们当初开始编写的仅仅是一个客户端应用，随后才意识到需要借助同构 JavaScript 提供的一些便利，比如优化页面加载速度。这种方式存在的问题是，客户端应用的实现通常是与浏览器环境紧密相连的，这使得将应用转移到服务器端的过程变得复杂了。这并不是说从服务器端开始编写就能免受特定环境问题的影响，但这样做确实可以保证我们可以从一个请求 / 响应生命周期的思维方式来开始编写，这正是服务器端应用所需要的。真正的优势是，我们不需要在现成的代码库上有所投入，因此我们可以从头开始编写！

6.1　提供HTML模板

在构建任何抽象或者定义 API 之前，为了熟悉服务器端的请求 / 响应生命周期，我们首先提供一份基于模板的 HTML 文档。这里选用 Mozilla（https://www.mozilla.org）公司提供的 Nunjucks（https://mozilla.github.io/nunjucks/）模板引擎。在之后的示例中，我们将一直使用这种模板语法。你可以通过如下方式安装 Nunjucks：

```
$ npm install nunjucks –save
```

安装 Nunjucks 后，就可以创建模板 src/index.html，其内容如例 6-1 所示。

例 6-1　使用 Nunjucks 编写 HTML 文档模板

```
<head>
  <meta charset="utf-8">
  <title>
    And the man in the suit has just bought a new car
    From the profit he's made on your dreams
  </title>
</head>
<body>
  <p>hello {{fname}} {{lname}}</p>
</body>
</html>
```

在模板上下文中，Nunjucks 使用双重花括号来渲染变量。接下来，修改 ./src/index.js 文件中的内容，让用户在浏览器中打开 localhost:8000/hello 时能够返回一个编译出姓氏和名字的模板。编辑后的文件内容如例 6-2 所示。

例 6-2　提供一份 HTML 文档（./src/index.js）

```
import Hapi from 'hapi';
import nunjucks from 'nunjucks';

// 配置Nunjucks以便从dist目录中读取内容
nunjucks.configure('./dist');

// 创建一个服务器,并配置主机名与端口
const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

// 添加路由
server.route({
  method: 'GET',
  path:'/hello',
  handler: function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', {
      fname: 'Rick', lname: 'Sanchez'
    }, function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
});

// 启动服务器
server.start();
```

我们对第 5 章中的代码示例作出了许多修改。现在来逐项分解并讨论这些内容。首先导入了 `nunjucks`。接下来对 `nunjucks` 进行了配置（https://mozilla.github.io/nunjucks/api.

html#configure)，让其从 ./dist 目录中读取内容。最后，使用 `nunjucks` 读取模板文件 ./dist/index.html，并使用上下文变量 `{ fname: Rick, lname: Sanchez }` 进行编译，编译结果返回一个 HTML 字符串，并将此作为服务器端的回复。

这段代码看起来很棒，但如果在终端运行 `gulp`，并尝试在浏览器中打开 localhost:8000/hello，你会发现服务器端只返回了一个空的 `<body>` 标签。为什么不能正常运行呢？如果你还记得的话，我们是在 ./src 目录中创建的模板，但却配置 `nunjucks` 让其从 ./dist 目录中读取内容（这确实是我们想要的，因为 ./dist 目录包含了应用的发行版，而 ./src 目录包含了应用的源代码）。那么应该如何修复这个问题呢？只需要修改构建流程并复制模板文件即可。修改 gulpfile.js 文件，添加一项复制任务并编辑原来的监听任务和默认任务，如例 6-3 所示。

例 6-3　将模板从 src 目录复制到 dist 目录（修改 gulpfile.js）

```
// 出于简洁的目的，省略部分原有代码

gulp.task('copy', function () {
  return gulp.src('src/**/*.html')
    .pipe(gulp.dest('dist'));
});

gulp.task('watch', function () {
  gulp.watch('src/**/*.js', ['compile']);
  gulp.watch('src/**/*.html', ['copy']);
});

// 出于简洁的目的，省略部分原有代码

gulp.task('default', function (callback) {
  sequence(['compile', 'watch', 'copy'], 'start', callback);
});
```

如果现在在终端运行 `gulp`，并在浏览器中打开 localhost:8000/hello，我们应该能够成功地看到“hello Rick Sanchez”了。

这个结果虽然相当不错，但并不是动态的。如果想要改变 `body` 响应中的姓名，该怎么做呢？我们需要将参数传递给服务器，这（在概念上）与向函数传递参数是类似的。

6.2　使用路径参数与查询参数

应用经常需要提供动态的内容。服务器端决定了这部分内容，方式包括路径参数、查询参数，有时候还会用到会话 cookie。在上一节中，欢迎信息中的姓名被硬编码在路由处理器中，但没有理由不让姓名的值由路径参数决定。 为了将路径参数传递到路由处理器中，需要修改路由中的 `path` 属性，如例 6-4 所示。

例 6-4　在 ./src/index.js 文件中添加路径参数

```
server.route({
  method: 'GET',
  path:'/hello/{fname}/{lname}',
  handler: function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', {
      fname: 'Rick', lname: 'Sanchez'
    }, function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
});
```

通过这次修改，路由现在能够匹配 localhost:8000/hello/morty/smith 和 localhost:8000/hello/jerry/smith 这样的 URI 了。URI 路径中提供的新值被称为**路径参数**（https://hapijs.com/api#path-parameters），并将成为请求对象的一部分，可以通过 request.params.fname 和 request.params.lname 获取这两个参数。这些值可以传递到模板上下文中，如例 6-5 所示。

例 6-5　访问路径参数

```
server.route({
  method: 'GET',
  path:'/hello/{fname}/{lname}',
  handler: function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', {
      fname: request.params.fname,
      lname: request.params.lname
    }, function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
});
```

如果现在在浏览器中加载 localhost:8000/hello/jerry/smith，应该能在响应体中看到路径参数了。

通过 URI 传值的另一种方式是使用**查询参数**（https://en.wikipedia.org/wiki/Query_string），如 localhost:8000/hello?fname=morty&lname=smith 和 localhost:8000/hello?fname=jerry&lname=smith。要想支持查询参数的使用，可以修改路由，如例 6-6 所示。

例 6-6　访问查询参数

```
server.route({
  method: 'GET',
  path:'/hello',
  handler: function (request, reply) {
    // 读取模板并使用上下文对象进行编译
```

```
    nunjucks.render('index.html', {
      fname: request.query.fname,
      lname: request.query.lname
    }, function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
});
```

这两种方式都可以在路由中获取动态值，并影响输出的动态内容。你可以同时使用这两种选项，并提供适当的默认值，以创建一个更加灵活的路由处理器，如例 6-7 所示。

例 6-7　同时访问路径参数与查询参数

```
function getName(request) {
  // 默认值
  let name = {
    fname: 'Rick',
    lname: 'Sanchez'
  };
  // 拆分路径参数
  let nameParts = request.params.name ? request.params.name.split('/') : [];

  // 优先顺序
  // (1) 路径参数
  // (2) 查询参数
  // (3) 默认值
  name.fname = (nameParts[0] || request.query.fname) ||
    name.fname;
  name.lname = (nameParts[1] || request.query.lname) ||
    name.lname;

  return name;
}

// 添加一条路由
server.route({
  method: 'GET',
  path:'/hello/{name*}',
  handler: function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', getName(request), function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
});
```

上述这些示例是为了简化而故意编造的，目的是让我们可以将关注点集中在概念上，但在现实世界中，路径参数与查询参数经常用于调用服务或者查询数据库。

第 8 章将介绍更多关于路由的细节，其中包括使用 call（https://github.com/hapijs/call）来创建同构路由，这是 hapi 使用的一个 HTTP 路由。

6.3　小结

在本章中，我们学习了如何基于渲染动态内容的模板来提供 HTML 文档，还进一步熟悉了请求 / 响应的生命周期以及其中的某些属性，如 `request.params` 和 `request.query`。这些知识将会贯穿本书余下的部分，并用于构建应用。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.2.x"` 命令，可以安装本章的完整代码示例。

第 7 章

设计应用架构

Jason Strimpel

如果你已经从事前端开发很长时间了，那么你可能还记得 MV* 框架大量涌现之前的那段日子。与那时相比，现在的 Web 开发更加复杂，也进步不少。如果在业界的时间再长一些，你大概还能想起 jQuery 出现之前的时光。对我而言，那段记忆充满了挫败感，我需要在一个长达 4000 行，并且包含各种各样的函数和“类”的 JavaScript 文件中进行调试，而且这个文件还引用了另一个有 4000 行代码的 JavaScript 文件。即使想不起这些日子，你应该也碰到过这样难以跟踪、令人沮丧的代码。

在大多数情况下，挫败感的源头是因为缺乏形式、结构，而这正是良好架构的基础。正如 James Coplien 和 Gertrud Bjørnvig 在 *Lean Architecture: For Agile Software Development*（Wiley）中提到的那样，“无须考虑事物是由什么构成的，将形式视为事物的一种基本形状或排列，而结构则是形式的物化”。在我们的示例中，形式的一个例子就是 `Application` 类，它负责接受路由定义。而结构的一个例子则是使用 `export` 和 `import` 的 ES6 模块格式。

另一个被过度使用的架构组件是**抽象**。合理使用抽象的关键在于只在必要时使用，因为隐藏细节会让代码变得难以跟踪和读取。否则，你会遇到数不清的包裹层函数和令人费解的实现，应用将会随着时间的推移而变得非常脆弱。在开发同构 JavaScript 应用的过程中，我们将会利用抽象来解决一些经常遇到的问题，比如 cookie 的获取和设置，但要注意不能滥用抽象。

在早期，重视形式与结构可以降低意外结果出现的可能性，并提高代码的可维护性。对抽象的合理怀疑可以确保我们的应用能够经受住时间的考验（至少五年）。

7.1 理解问题

我们知道自己目前正在创建的是同构 JavaScript 应用，但“架构”的含义到底是什么呢？为了回答这个问题，首先要定义需要解决的问题。我们的目标是，在 Web 浏览器中高效地提供一个兼容 SEO 的 UI。在这个目标中，效率与应用可以同时在客户端和浏览器端运行紧密相关，因为这样应用才能利用这两种环境带来的便利。既然应用必须在客户端和服务器端运行，那么我们就必须考虑如何在抽象环境的同时，避免引入不必要的复杂度。那么从哪里开始讨论呢？从用户请求开始，讨论连接一切（http://radar.oreilly.com/2015/06/the-power-of-connection.html）的创新方式：URL。

7.2 响应用户请求

URL 是连接用户和应用的桥梁。应用使用 URL 建立特定资源的映射关系，并根据应用逻辑将资源返回给用户。URL 可以让 Web 运转，因此也是向应用添加结构的一个合适入口。在上一章的示例中，我们使用了 hapi 的 `server.route` API 向应用中添加路由来响应用户请求。这种实现仅仅适用于在服务器端运行且将 hapi 作为应用服务器的应用。但在我们的示例中，我们希望应用不仅能在服务器端运行，还可以在客户端运行，因此直接引用 hapi 还不够。此外，在应用各处直接使用 hapi 的 API 会导致应用代码和 hapi 强耦合，进而导致你以后很难在需要的情况下替换 hapi 的框架。

抽象

人们有时候会将抽象考虑得太过长远。例如，仅仅为了方便以后替换库，就在应用中围绕某个库创建 API 包裹层，这就不算是一个好的理由。因为你想要解决的是一个尚未发生，而且可能永远不会发生的问题，所以现在将时间花费在这里并不明智。抽象应该能够立刻（或者即将）提供价值。我们应该使用良好的形式和结构来确保软件寿命，而不是过多、过早地进行抽象。

7.2.1 创建Application类

要想在响应用户请求时提供结构，第一步是创建一个 `Application` 类，以便在应用范围内重用。创建这个类的目的是减少模板代码，并提供接口最终供客户端与服务器端共同使用。

在定义用户路由的示例中，需要在 ./src/index.js 文件中用到 `Application` 类的接口，如例 7-1 所示。

例 7-1　使用 Application 类

```
import Hapi from 'hapi';
import nunjucks from 'nunjucks';
import Application from './lib';
```

```
// 配置Nunjucks以便从dist目录中读取内容
nunjucks.configure('./dist');

// 创建一个服务器,并配置主机名与端口
const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

function getName(request) {
  // 出于简洁的目的,省略函数体代码
}

const application = new Application({
  // 响应来自http://localhost:8000/的请求
  '/': function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', getName(request), function (err, html) {
      // 使用HTML内容响应请求
      reply(html);
    });
  }
}, {
  server: server
});

application.start();
```

> 前面的内容谈到了要正确地进行抽象。在这个示例中，对服务器实例化的细节进行抽象并不会带来好处，因此我们没有改动。如果配置内容的增长让应用的文件难以跟踪，或者需要注册大量的 hapi 插件（http://hapijs.com/plugins），我们以后可能会将这些细节分离到一个独立的模块中。

如果看到例 7-1 后，你的第一反应是“我看不出这样做有什么好处”，别担心，这种反应是正常的。我们实现一个 `Application` 类当然不能仅仅用来支撑这一部分代码，因为除了封装实现细节之外，这样做没有提供任何好处。在示例中，我们建立了一个基础，并在这个基础之上进行完善，这样做的好处将会在第二部分的教程中逐渐显露出来。明确了这个示例的用意之后，来继续关注其实现。在 /src/lib/index.js 文件中定义 `Application` 类，如例 7-2 所示。

例 7-2　`Application` 类

```
export default class Application {

  constructor(routes, options) {
    this.server = options.server;
    this.registerRoutes(routes);
  }
```

```
  registerRoutes(routes) {
    for (let path in routes) {
      this.addRoute(path, routes[path]);
    }
  }

  addRoute(path, handler) {
    this.server.route({
      path: path,
      method: 'GET',
      handler: handler
    });
  }

  start() {
    this.server.start();
  }

}
```

现在，我们已经封装（façade）了一个基本的应用，并且最终还要修改客户端实现。这是一个很好的开始，但正如前面提到的那样，除了为随后传输到客户端做准备之外，我们还没有真正为应用添加任何东西。为了在现阶段增加一些价值，我们需要适当减少路由定义的代码，并为响应用户请求的逻辑添加更多的结构。

7.2.2 创建控制器

通过创建一种通用的方式来响应 URL 请求，我们可以进一步改进结构并减少模板代码。为了实现这一点，我们需要创建一个接口，让开发人员可以针对这个接口进行编码。正如在例 7-2 中建立的应用结构那样，这种做法在现阶段的示例中不会有什么好处，但我们可以在此基础上继续构建。

在 Struts、Ruby on Rails、ASP.Net 等框架中，控制器有供框架调用的操作方法。控制器及其操作与路由表中的路径存在对应关系。这些操作方法包含了分别处理传入请求和响应的业务逻辑。在示例中，我们想要响应一个 UI，即一份 HTML 静荷数据。了解了这些以后，我们开始定义一个基本的接口，如例 7-3 所示（./src/lib/controller.js）。

例 7-3 `Controller` 类

```
export default class Controller {

  constructor(context) {
    this.context = context;
  }

  index(application, request, reply, callback) {
    callback(null);
  }
```

```
    toString(callback) {
      callback(null, 'success');
    }

  }
```

`constructor` 方法创建了 `Controller` 类的一个实例。`context` 参数包含了与路由相关的元数据，比如路径参数和查询参数。当控制器实例需要在操作响应请求后持续存在时，这些数据就能够在客户端派上用场。

`index` 方法是一个控制器实例的默认操作，该方法接收 4 个参数。

(1) `application` 是定义路由的 `Application` 类对象的一个引用。在将来需要访问应用层面的方法和属性时，这个参数十分有用。

(2) `request` 即 hapi 的 `request` 对象。这个参数可以用于请求层面的操作，比如读取 HTTP 头部信息或 cookie 的值。这个对象未来将会被规范化，以便在客户端和服务器端可以调用相同的函数方法。

(3) `reply` 即 hapi 的 `reply` 对象。这个参数可以用来重定向请求，如 `reply.redirect(`*`some/url`*`)`。这个对象未来也将会被规范化，以便可以在客户端和服务器端调用相同的函数方法。

(4) `callback` 是一个**遵循 Node 风格的回调函数**（http://fredkschott.com/post/2014/03/understanding-error-first-callbacks-in-node-js/），用于处理异步控制流。如果函数接收的第一个参数是 `null`，那么负责调用操作方法的控制器就会进入请求 / 响应的生命周期中。如果函数接收的第一个参数是 `Error`（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error），那么应用就会响应一个错误（后文将会详细讨论错误响应）。

如果操作方法 `callback` 成功执行且没有出现错误，那么应用框架随后就会调用 `toString` 方法。一个成功的 `callback` 的第二个参数应该是需要被渲染的字符串。

7.2.3 构造控制器实例

定义好响应资源请求的合约后，就可以将更多与定义路由相关的逻辑移到应用框架中。如果你还记得的话，./src/index.js 文件中就使用了内联函数来包含路由的定义（如例 7-4 所示）。

例 7-4　内联路由处理器

```
const application = new Application({
  // 响应来自http://localhost:8000/的请求
  '/': function (request, reply) {
    // 读取模板并使用上下文对象进行编译
    nunjucks.render('index.html', getName(request), function (err, html) {
```

```
      // 使用HTML内容响应请求
      reply(html);
    });
  }
}, {
  server: server
});
```

定义好一个基础的控制器后，可以将上述示例修改为例 7-5 所示的版本。

例 7-5　使用 Controller 类

```
import Hapi from 'hapi';
import Application from './lib';
import Controller from './lib/controller'

const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

const application = new Application({
  '/': Controller
}, {
  server: server
});

application.start();
```

现在代码看起来好多了。我们成功地移除了渲染和响应的实现细节，从而让应用代码更加容易阅读。我们看一眼就可以很快知道，`Controller` 类将会负责响应来自 http://localhost:8000/ 的请求。这才是一个真正的艺术品。但很可惜，代码现在还不能正常工作。我们需要在 src/lib/index.js 文件中编写逻辑代码，这个文件将会创建一个通过路由处理器响应请求的实例。需要修改的地方是 `Application` 类的 `addRoute` 方法，如例 7-6 所示。示例中的代码创建了一个处理器，这个处理器又创建了一个控制器实例，并遵循了控制器生命周期的合约。

例 7-6　Application 类的 addRoute 方法

```
addRoute(path, Controller) {
  this.server.route({
    path: path,
    method: 'GET',
    handler: (request, reply) => {
      const controller = new Controller({
        query: request.query,
        params: request.params
      });

      controller.index(this, request, reply, (err) => {
        if (err) {
```

```
          return reply(err);
        }

        controller.toString((err, html) => {
          if (err) {
            return reply(err);
          }

          reply(html);
        });
      });
    }
  });
}
```

> 这个示例引入了一些新的语法。如果对**箭头函数**（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions）并不熟悉的话，可能会产生一些疑惑。在这个示例中，使用箭头函数的目的是实现 `this` 的绑定，从而无须额外创建 `self` 或者 `that` 这样的变量，或者显式地使用 bind（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind）函数来设定上下文对象。

如果现在在浏览器中打开 http://localhost:8000/，你应该可以看到“success”提示信息。这个结果符合预期，但还不是我们真正想要的。

7.2.4　拓展控制器

在之前的路由处理器中，我们将 ./src/index.html 的文件内容传递到 Nunjucks 中进行编译，并提供一个 `context` 对象，最终使用模板函数返回的字符串内容进行响应。接下来看看新架构的用法。例 7-7 展示了一个拓展后的 `Controller` 类（代码保存在 /src/HelloController.js 中）。

例 7-7　拓展基类控制器

```
import Controller from './lib/controller';
import nunjucks from 'nunjucks';

// 配置Nunjucks以便从dist目录中读取内容
nunjucks.configure('./dist');

function getName(context) {
  // 出于简洁的目的,省略函数体代码
}

export default class HelloController extends Controller {

  toString(callback) {
    // 读取模板并使用上下文对象进行编译
```

```
    nunjucks.render('index.html', getName(this.context), (err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }

}
```

这个函数的功能和例 7-4 所示功能基本相同，但我们在这个控制器中封装了负责响应 http://localhost:8000/ 的业务逻辑。如果有必要，也可以让它成为应用的基类控制器，并创建一些约定，以根据路由解析对应的模板文件。例如，可以让 `request.uri` 成为 `context` 对象的一部分，并根据 `request.uri.path` 在 ./dist 目录的相应位置来定位模板文件。我们并不打算在此实现这种基于约定的方案，但这里想说明的关键点是，你可以将应用中特定的通用代码放在基类控制器中，以促进代码重用。

接下来需要修改 ./src/index.js 文件来使用新的控制器，并修改路由路径，以接受可选的路径参数，如例 7-8 所示。

例 7-8　接受路径参数

```
import Hapi from 'hapi';
import Application from './lib';
import HelloController from './hello-controller';

const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

const application = new Application({
  '/hello/{name*}': HelloController
}, {
  server: server
});

application.start();
```

现在，如果在浏览器中打开 http://localhost:8000/hello/{fname}/{lname}，应该能够看到想要的欢迎信息了。

7.2.5　改进响应流

在将代码移植到客户端之前，还需要修改最后一处实现。在控制器中更换原有的读取并创建 HTML 响应的方式，取而代之的，创建一个返回页面模板的 API，并将结果注入控制器的 `toString` 回调函数中。新版本的 ./src/HelloController.js 文件如例 7-9 所示。

例 7-9　创建内联模板

```
import Controller from './lib/controller';
import nunjucks from 'nunjucks';

function getName(context) {
  // 出于简洁的目的,省略函数体代码
}

export default class HelloController extends Controller {

  toString(callback) {
    nunjucks.renderString('<p>hello {{fname}} {{lname}}</p>', getName(this.context),
(err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }

}
```

这样做的原因是，当应用代码转换为客户端 SPA 之后，路由只会针对每条路由返回对应的 HTML 内容，而不是一份完整的文档。这样做还具有性能方面的优势。例如，我们或许可以在服务器端限制文件系统 I/O 的数量。而在客户端，可以创建一个包含不会重复渲染的页眉和页脚的基本布局，以便在重定向时不需要重新解析 `<script>` 标签。从应用的角度来看，需要在 ./src/index.js 文件中作出这些修改（如例 7-10 所示）。

例 7-10　定义应用的 HTML 文档

```
import Hapi from 'hapi';
import Application from './lib';
import HelloController from './hello-controller';
import nunjucks from 'nunjucks';

// 配置Nunjucks以便从dist目录中读取内容
nunjucks.configure('./dist');

const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

const application = new Application({
  '/hello/{name*}': HelloController
}, {
  server: server,
  document: function (application, controller, request, reply, body, callback) {
    nunjucks.render('./index.html', { body: body }, (err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }
```

```
});

application.start();
```

这些修改允许我们将控制器的 `toString` 回调函数值和 `body` 参数注入到模板内容中，并且可以脱离具体框架的选择。现在对应用框架代码进行修改。`addRoute` 方法的最终版本如例 7-11 所示。

例 7-11　响应资源请求（`Application` 类的 `addRoute` 方法）

```
addRoute(path, Controller) {
  this.server.route({
    path: path,
    method: 'GET',
    handler: (request, reply) => {
      const controller = new Controller({
        query: request.query,
        params: request.params
      });

      controller.index(this, request, reply, (err) => {
        if (err) {
          return reply(err);
        }

        controller.toString((err, html) => {
          if (err) {
            return reply(err);
          }

          this.document(this, controller, request, reply, html,
           function (err, html) {
           if (err) {
             return reply(err);
           }

           reply(html);
          });
        });
      });
    }
  });
}
```


应用框架现在负责组合 HTML 文档响应，但 HTML 字符串构造的实现细节则留给应用开发者自行决定。最后，我们需要修改模板，如例 7-12 所示（./src/index.html）。

例 7-12　为文档主体添加出口 [2]

```
<head>
  <meta charset="utf-8">
  <title>
```

注 2：在新版本的 Nunjucks 中，下述代码中的 `{{body}}` 内容可能会被自动转义。要想禁止自动转义，可以使用 `{{body | safe}}` 或 `nunjucks.configure({autoescape: false})` 进行全局配置。——译者注

```
      And the man in the suit has just bought a new car
      From the profit he's made on your dreams
    </title>
  </head>
  <body>
    {{body}}
  </body>
  </html>
```

如果在浏览器中打开 http://localhost:8000/hello/{fname}/{lname}，你现在应该能够看到“hello word”了。

> ES6 的模板字符串 API（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals）也可以用于向字符串中嵌入表达式。之所以没有使用这个 API，是因为我们是从文件系统中读取内容的。要想将文件内容转换为模板字符串，需要使用 `eval`（用法为 `eval(templateStr);`），但这种做法会存在某些安全风险。

图 7-1 展示了完整的请求 / 响应生命周期。

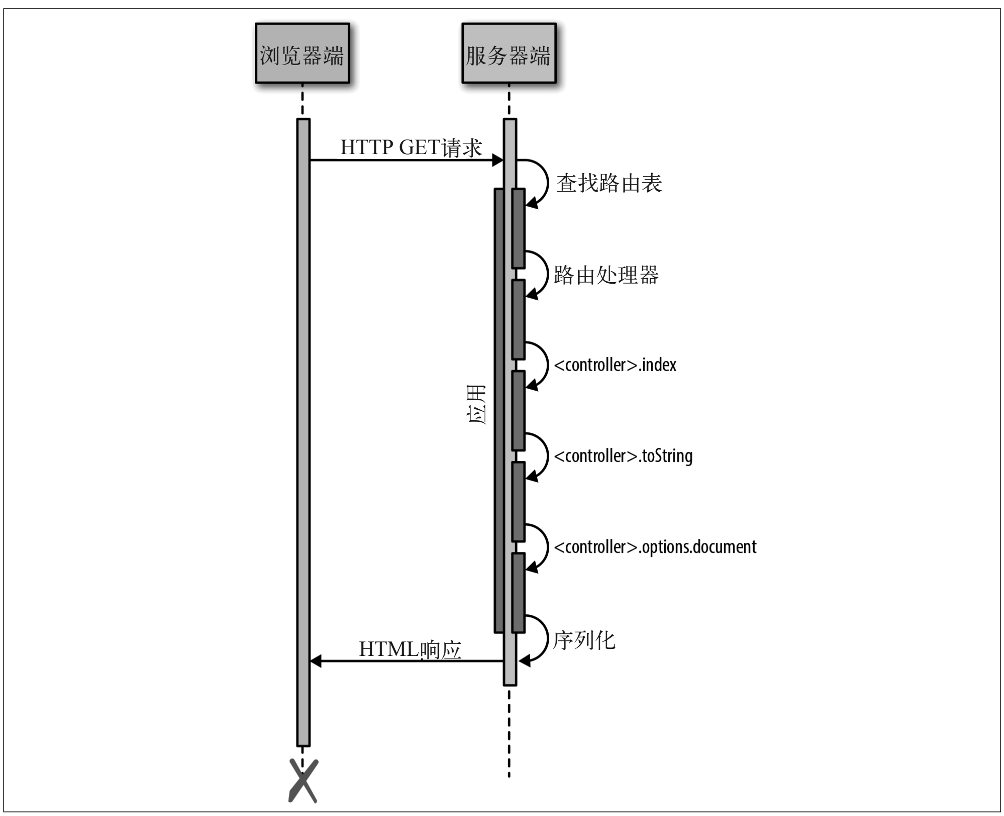


图 7-1：请求 / 响应生命周期

7.3　小结

我们在本章中为后续开发建立了一个坚实的基础，通过在应用中添加形式与结构完成了这项任务。我们在应用和框架之间定义了明确的约定，用于响应资源请求。这些特意的设计将会帮助我们在未来更加便捷地更换应用中的技术栈，以便轻松应对大量的更改，同时也有助于尝试使用新技术。更重要的是，这样做可以确保应用的稳定性。在下一章中，我们将开始做一些真正有趣的事情，将应用和框架移植到客户端。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.3.x"` 命令，可以安装本章的完整代码示例。

第 8 章

将应用传输到客户端

Jason Strimpel

到目前为止，我们一直专注于为应用打造坚实的基础。本章会将原本只能在服务器端运行的应用修改为在客户端运行，开始从我们的精心策划中获得便利。如果之前完成的工作是有效的话，那么完成这项任务应该相当轻松；完成这一步骤后，我们就将拥有一个功能完善的同构 JavaScript 应用核心。不过，在开始将应用移植到客户端之前，需要对构建过程进行一些补充，并修改应用的结构。

8.1　打包应用的客户端版本

为了让应用在客户端运行，首先需要为应用打包一个文件，其中包含整个应用的源代码。这个文件将被引入 ./src/index.html，作为服务器端提供的首屏响应内容。

> 如果你的应用规模比较大，那么可能需要将代码分割成多个小文件，以改善首屏加载的体验。

8.1.1　选择打包库

谈到客户端应用打包，目前社区中使用的主流打包库有两款：Browserify（http://browserify.org）和 Webpack（http://webpack.github.io/）。

异步模块定义（Asynchronous Module Definition，AMD）

第三款打包库称为 RequireJS Optimizer（http://requirejs.org/docs/optimization.html），这个库借助了 RequireJS，而 RequireJS 遵循了 AMD 模式（https://github.com/amdjs/amdjs-api/blob/master/AMD.md）。AMD 是指定模块定义机制的一个 API，使得模块及其相关的依赖可以异步加载。然而，业界现在更倾向于同步模块加载的模式，如 CommonJS（http://wiki.commonjs.org/wiki/CommonJS），因此我们不会讨论这部分内容。

通过使用 `require` 语法来引入依赖，Browserify 可以让你像编写 Node 应用那样开发客户端应用。它还提供了某些 Node 核心库的客户端版本，让你可以在客户端引入它们，从而像在服务器端那样使用这些 API。

Webpack 可以为客户端打包所有的资源类型，包括 CSS、AMD、SASS、图片、CoffeeScript 等。它包含了一些内建的插件，并且支持代码拆分的概念，这使得你可以轻松地将应用分割成多个小文件，以避免在初始加载时加载整个应用的内容。

这两个库都是很不错的选择，使用哪一个都可以完成相同的结果。只不过它们的实现方式存在一些差异。本书选择使用 Browserify，因为它在我们的用例中不需要进行太复杂的配置，更加容易上手。

8.1.2　创建打包任务

在这一节中，我们将为客户端创建第一份打包。打包任务非常容易执行，但需要进行一些初始化配置。这个过程的第一步是安装一些新的构建工具，我们从 `browserify` 模块开始：

```
$ npm install browserify --save-dev
```

在接下来的过程中，我们将在 gulpfile.js 文件中利用 `browserify` 创建一项任务，即构建应用打包。我们还需要安装 `babelify` 模块：

```
$ npm install babelify --save-dev
```

Babelify（https://github.com/babel/babelify）是 Browserify 的转换器（https://github.com/substack/browserify-handbook#transforms），用于将源代码从 ES6 转换为 ES5，就像我们的 `compile` 编译任务那样。这在现在看来似乎有点多余，但如果未来需要添加其他的转换过程，比如 `brfs`（https://github.com/substack/brfs），那么就有必要对源代码进行打包并使用转换器，而不是对已经编译好的发行版本进行打包。由于需要将传统的文本流从 Browserify 管道传递到 Gulp，因此还需要安装 `vinyl-source-stream`：

```
$ npm install vinyl-source-stream --save-dev
```

Node 中的流

在 Node 中，流（https://nodejs.org/api/stream.html）是一个抽象的接口，可以由不同的对象实现。举例来说，对 HTTP 服务器的一个请求就是一个流，如 `stdout` 就是一个流。流是可读、可写，或者同时可读写的。所有的流都是 EventEmitter 的实例。

接下来要为 Browserify 提供一些指令，以便需要时用它打包客户端专用的实现。在 package.json 文件中添加 `browser` 属性，如下所示：

```
{
  "browser": {
    "./src/index.js": "./src/index.client.js"
  }
}
```

当遇到特定文件 ./src/index.js 时，Browserify 应该知道要打包一个不同的文件 .src/index.client.js，这个文件中包含了客户端版本的实现。这种做法乍听起来有些奇怪，因为我们本来要编写的代码就是假定可以同时在客户端和服务器端运行的，但某些时候我们确实不能这样做（比如不能在客户端运行 hapi 服务器）。关键是要限制并隔离这些不经常改变的代码补丁，从而让日常开发不会因环境上下文切换而受到巨大影响。

最后一步是修改 gulpfile.js 文件。首先需要导入新安装的模块。

```
var browserify = require('browserify');
var source = require('vinyl-source-stream');
```

然后创建新的 `bundle` 打包任务。

```
gulp.task('bundle', function () {
  var b = browserify({
    entries: 'src/index.js',
    debug: true
  })
  .transform('babelify', { presets: ['es2015'] });

  return b.bundle()
    .pipe(source('build/application.js'))
    .pipe(gulp.dest('dist'));
});
```

这项任务会通过 `browserify` 运行 ./src/index.client.js 文件，并追踪所有发现的依赖。任务会创建一个单独的文件，并写入到 ./dist/build/application.js 中。接下来，将 `bundle` 任务添加到默认任务中。

```
gulp.task('default', function (callback) {
  sequence(['compile', 'watch', 'copy', 'bundle'], 'start', callback);
});
```

最后，需要修改 watch 任务，以便修改源代码后可以自动重新打包。

```
gulp.task('watch', function () {
  gulp.watch('src/**/*.js', ['compile', 'bundle'])
  gulp.watch('src/**/*.html', ['copy']);
});
```

就是这样！现在我们已经准备好创建打包文件了，不过需要按照之前在 package.json 文件中指定的那样，先添加客户端的实现。

8.1.3 添加客户端实现

在应用的入口点 ./src/index.js 文件中，我们实例化了一个 hapi 服务器。这仅仅是一种用于特定环境的代码，我们需要确保其不会在客户端中运行。在上一节中，我们已经看到了如何在 package.json 文件中通过 browser 属性为客户端和服务器端指定不同的实现，也定义了用于替换 ./src/index.js 的文件 ./src/index.client.js。我们的第一个目标是简单地将“hello browser”打印到浏览器控制台中。

```
console.log('hello browser');
```

现在需要在应用模板 ./src/index.html 文件中引入这个文件的链接地址，如例 8-1 所示。

例 8-1　在页面模板中引入应用打包文件

```
<html>
  <head>
    <meta charset="utf-8">
    <title>
      And the man in the suit has just bought a new car
      From the profit he's made on your dreams
    </title>
  </head>
  <body>
    {{body}}
  </body>
  <script type="text/javascript" src={{application}}></script>
  </html>
```

在 ./src/index.js 文件中将应用打包文件的路径作为属性值，并通过渲染上下文进行参数传递，如例 8-2 所示。

例 8-2　在模板渲染上下文中添加打包路径

```
const APP_FILE_PATH = '/application.js';
const application = new Application({
  '/hello/{name*}': HelloController
}, {
  server: server,
  document: function (application, controller, request, reply, body, callback) {
    nunjucks.render('./index.html', {
```

```
      body: body,
      application: APP_FILE_PATH
    }, (err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }
});
```

最后，在服务器代码 ./src/index.js 中添加一条路由来提供打包文件。

```
server.route({
  method: 'GET',
  path: APP_FILE_PATH,
  handler: (request, reply) => {
    reply.file('dist/build/application.js');
  }
});
```

现在，在终端执行 Gulp 默认任务并在浏览器中打开 http://localhost:8000/，应该能看到和以往一样的结果。但如果打开浏览器控制台，应该能看到“hello browser”字样的提示信息。如果你能看见这条信息，那么恭喜你——你提供了第一个应用包！虽然这个示例很简单，但理解其设置步骤将有助于实现接下来的同构工作。

8.2 响应用户请求

在上一章中，我们将 URL 作为让用户向应用发出请求的一种机制。以这个 Web 基石作为切入点，我们在服务器上建立了应用框架。我们接收传入的请求，并将请求映射到执行控制器操作的路由处理器中。这种做法用于构造客户端请求的响应。这个请求 / 响应的生命周期构成了应用框架的核心，而我们必须确保客户端也能够支持这种生命周期的协议，以便应用代码可以通过可预测的方式执行。

协议的第一部分是需要响应用户发起的请求（即 URL）。这在服务器端是一个 HTTP 的 `request` 对象。客户端中没有这个对象，但我们还是希望应用代码能够在客户端执行，以便利用 SPA 模型带来的性能优势。在客户端中，我们可能会响应用户点击的一个链接，并随之更新浏览器地址栏中的 URL。正是因为 URL 发生了变化，所以必须在客户端中进行响应，正如在服务器端响应 HTTP 请求一样。为了在客户端执行请求 / 响应生命周期，我们的基本思路是劫持点击操作，这些操作本来会正常地改变 URL，并发起一个 HTTP 请求以获取 HTML 文档，导致页面重新加载。此外，我们还希望能够确保浏览器历史正常保留，以便在浏览器中执行的前进或者后退操作能够如常进行。好在可以利用现有的原生接口 History API（https://developer.mozilla.org/en-US/docs/Web/API/History）来实现这些内容。

8.2.1　利用History API

在 History API 出现之前，SPA 在客户端中使用 **# 号片段**（https://en.wikipedia.org/wiki/Fragment_identifier）作为“页面”路由的一种解决方案。当 # 号片段发生变化时，会在浏览器历史中增添新的记录，而页面不会刷新，但这种方式不支持 SEO，因为 # 号片段不会作为 HTTP 请求的一部分被发送到服务器端。这是因为 # 号片段的设计目的仅仅是链接到文档中的某一部分。然而，History API 诞生的目的就是为了确保 URL 依然可以达到预期的目的，即在 SPA 和被搜索引擎正确索引的那些内容中**识别唯一的资源**。

History API 非常简单。它提供了一个 history 栈，你可以将状态对象、标题和 URL 添加到这个栈中。对我们而言，在路由表将 URL 映射到路由中时，我们只需要关心其中的两个方法和一个事件。

`History.replaceState`

　这个方法会修改 history 栈中最新的一项历史，可以用于向一个服务器端渲染的页面中添加状态对象。

`History.pushState`

　这个方法会向 history 栈中添加一个状态对象、标题（可选）和 URL（可选）。这个方法可以帮助我们存储 URL 状态，并提高客户端跳转的响应性。比如，渲染页面所需的所有数据都可以存储在这个状态对象中，这样一来，当用户跳转到之前渲染过的页面时，就可以对网络请求进行数据短路。

`PopStateEvent`

　当前活动历史项发生改变时会触发 `PopStateEvent`，比如当用户点击浏览器的后退按钮时。监听该事件可以用于触发客户端的路由跳转。

这些方法和事件可以在客户端为每一个通过 URL 标识的唯一资源触发路由请求，就像在服务器端使用的 HTTP `GET` 请求那样。

8.2.2　响应并调用History API

在本节中，我们会将 History API 整合到应用的核心代码当中，以实现客户端路由。在开始之前需要提醒你的是，我们将在本节中创建第一个真正的抽象。通常我会极力避免进行抽象，因为这会隐藏细节，从而混淆代码含义，让代码更加难以跟踪，结构变得脆弱。正如 James Coplien 所说，“抽象是魔鬼”。不过，抽象有时确实是必需的，这个示例就恰好属于这种情况，因为我们不可能在客户端运行一个服务器。

在 8.1.3 节中，我们创建了一个客户端打包文件，该文件在日志中记录了“hello browser”。这个打包文件的入口点是 ./src/index.js 文件，随后，我们在 package.json 中将入口点指向了客户端实现的版本，即 ./src/index.client.js，而 ./src/index.js 文件则成为了服务器端的入口点。

这个服务器端的入口点导入了应用的核心代码，即 ./src/lib/index.js，并启动了应用。我们需要在客户端实现中遵循相同的形式，如例 8-3 所示。

例 8-3　客户端打包入口点

```
import Application from './lib';
import HelloController from './HelloController';

const application = new Application({
  '/hello/{name*}': HelloController
}, {
  // target参数设置控制器响应的内容应该插入到哪个元素中
  // 值为元素的查找选择器(query selector)
  target: 'body'
});

application.start();
```

> 目前我们将关注点放在 History API 本身，暂时不考虑代码重用的问题。在完成客户端实现的最初版本之后，我们再考虑重用的问题。此外，我们在这一节中会跳过路由定义，并将在 8.3 节中进行探讨。

接下来需要实现客户端的 `Application` 类，并在其中封装 History API 的代码。但是，首先需要在 package.json 的 `browser` 属性中添加一个新的字段。

```
{
  "browser": {
    "./src/index.js": "./src/index.client.js",
    "./src/lib/index.js": "./src/lib/index.client.js"
  }
}
```

这是为了通知 Browserify 在打包时使用 ./src/lib/index.client.js 文件。现在我们可以开始在 ./src/lib/index.client.js 文件中实现 `Application` 类，如例 8-4 所示。

例 8-4　`Application` 类的待实现函数

```
export default class Application {

  navigate(url, push=true) {

  }

  start() {

  }

}
```

我们在本节中将以这种形式进行编码，从实现 `start` 方法开始。首先需要为 `History.pushState` 添加一个事件监听器（如例 8-5 所示）。

例 8-5　Application 类的 navigate 和 start 方法

```
navigate(url, push=true) {
  console.log(url);
}

start() {
  // 创建popstate事件监听
  this.popStateListener = window.addEventListener('popstate', (e) => {
    let { pathname, search} = window.location;
    let url = `${pathname}${search}`;
    this.navigate(url, false);
  });
}
```

目前来看，这个事件监听器只是简单地将当前的 URL 打印出来，让我们能够确认它是正常工作的。在 8.3 节中，我们会将它和客户端路由联系起来，执行一个匹配 URL 的路由处理器。

接下来，需要实现一个选择性加入的点击事件处理器。当用户点击一个带有 href 属性的标签或者某些选择性加入的提供所需数据的元素时，这个处理器可以用来执行路由处理器。这种选择应该是声明式的和不显眼的，以便应用框架可以轻松地进行事件监听，而不会影响到应用的其他部分。要实现这一目的，使用 data-* 属性（https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/data-*）是一个不错的方法。我们可以使用这个接口来定义自己的 data-* 属性，并利用这个属性来检测应在应用框架中处理的点击事件（如例 8-6 所示）。

例 8-6　Application 类的 start 方法中的事件监听器

```
start() {
  // 创建popstate事件监听器
  this.popStateListener = window.addEventListener('popstate', (e) => {
    // 出于简洁的目的,省略函数体代码
  });

  // 如果符合执行条件,就创建点击事件监听器,并将跳转的工作转交给navigate方法
  this.clickListener = document.addEventListener('click', (e) => {
    let { target } = e;
    let identifier = target.dataset.navigate;
    let href = target.getAttribute('href');

    if (identifier !== undefined) {
      // 用户点击链接时,需要避免浏览器默认行为(即加载一个新的HTML文档)
      if (href) {
        e.preventDefault();
      }

      // 如果identifier存在,则使用identifier进行跳转,否则使用href
      this.navigate(identifier || href);
    }
  });
}
```

上述代码在 document 对象中放置了一个事件监听器，并根据自定义数据属性 data-navigate 进行筛选。如果符合执行条件，那么就调用一个被称为 navigate 的新方法（尚未实现）。接下来，我们来实现点击事件处理器引用到的这个 navigate 方法（如例 8-7 所示）。

例 8-7　Application 类的 navigate 和 history.pushState 方法

```
navigate(url, push=true) {
  // 如果浏览器不支持History API,则直接设置location属性并返回
  if (!history.pushState) {
    window.location = url;
    return;
  }

  console.log(url);

  // 只有在push参数为true时才添加到history栈中
  if (push) {
    history.pushState({}, null, url);
  }
}
```

这个 navigate 方法是用于为匹配路由表的 URL 执行路由处理器的另一个占位方法。目前只需要将一个空的状态对象和一个 URL 添加到 history 栈即可，以确保这个方法可以正常工作。

现在已经实现了桩函数中的内容，还需要修改 /{name*} 路由对应的模板，将一些链接放进其中来测试新方法。目前的模板被硬编码在 HelloController 类中，以字符串形式存在（在 ./src/hello-controller.js 中定义），因为这个模板很简单，所以我们可以避免读取文件系统的开销。然而，既然已经打算扩展模板，现在似乎就是将模板移动到一个单独文件的好时机，将模板放在 ./src/hello.html 文件中（如例 8-8 所示）。

例 8-8　HelloController 类的模板

```
<p>hello {{fname}} {{lname}}</p>
<ul>
  <li><a href="/mortimer/smith" data-navigate>Mortimer Smith</a></li>
  <li><a href="/bird/person" data-navigate>Bird Person</a></li>
  <li><a href="/revolio/clockberg" data-navigate>Revolio Clockberg</a></li>
</ul>
```

最后需要修改 HelloController 类（如例 7-7 所示），让其从文件系统中读取模板，如例 8-9 所示。

例 8-9　HelloController 类的 toString 方法

```
toString(callback) {

  nunjucks.render('hello.html', getName(this.context), (err, html) => {
    if (err) {
      return callback(err, null);
    }
```

```
    callback(null, html);
  });
}
```

如果在终端中执行 gulp，并在浏览器中打开 http://localhost:8000/hello/{fname}/{lname}，那么你应该可以看到带有链接的新页面。当点击链接时，你应该可以看到浏览器地址栏发生了变化，并且在控制台中打印出了日志语句。当使用浏览器的前进和后退功能时，也应该能够看到相同的行为。现在我们已经挂钩（hook）到了浏览器历史当中！

从表面上看，利用 History API 的这些待实现函数显得微不足道，但它们将会成为应用的资源请求接口，就像在服务器端进行的 HTTP GET 请求那样。URL 正是服务器端和客户端实现的共同因素。URL 就像函数的签名那样，用于在路由表中匹配路由。在服务器端，路由表是 hapi 的一部分。虽然我们不能在客户端中运行 hapi，但我们应该使用相同的路由规则，以便路由可以通过同一种算法来匹配和运用。在下一节中，我们将探讨如何实现这一点。

8.3 客户端路由

在上一节中，当点击链接或者通过浏览器历史进行前进或者后退的跳转时，你可能已经注意到，屏幕上的欢迎信息没有随着路径参数的改变而发生变化。这是因为我们没有执行控制器操作并渲染响应。为了做到这一点，我们需要一个客户端路由器，这个路由器使用的路由匹配算法与 hapi 一致。好在 hapi 将其 HTTP 路由器模块化了，并称之为 call（https://www.npmjs.com/package/call），而且由于 Browserify 本身就是设计用于在客户端运行 Node 模块的，因此我们正好可以利用这个模块！但首先需要安装它。

```
$ npm install call --save
```

接下来需要将 call 模块导入到应用框架的客户端源代码 ./src/lib/index.client.js 中，并编写 constructor 和 registerRoutes 方法的客户端实现，如例 8-10 所示。

例 8-10　在 Application 类中使用 HTTP 路由器 call

```
import Call from 'call';

export default class Application {

  constructor(routes, options) {
    // 为控制器保存路由为查找表
    this.routes = routes;
    this.options = options;
    // 创建一个call的路由器实例
    this.router = new Call.Router();
    this.registerRoutes(routes);
  }

  registerRoutes(routes) {
```

```
    // 遍历传入的路由,并将这些路由添加到call的路由器实例中
    for (let path in routes) {
      this.router.add({
        path: path,
        method: 'get'
      });
    }
  }

  navigate(url, push=true) {
    // 出于简洁的目的,省略函数体代码
  }

  start() {
    // 出于简洁的目的,省略函数体代码
  }

}
```

在 constructor 和 registerRoutes 方法中，我们使用了 call 模块来创建路由器，并用来注册应用路由。这些路由定义会由 navigate 方法使用，通过在 constructor 方法中设置的 this.routes 属性将 URL 匹配到控制器（如例 8-11 所示）。

例 8-11　在 Application 类的 navigate 方法中匹配路由

```
navigate(url, push=true) {
  // 如果浏览器不支持History API,则直接设置location属性并返回
  if (!history.pushState) {
    window.location = url;
    return;
  }

  // 分割路径并搜索字符串
  let urlParts = url.split('?');
  // 解构urlParts数组
  let [path, search] = urlParts;
  // 判断URL路径是否匹配路由器中的路由
  let match = this.router.route('get', path);
  // 解构路由的路径和参数
  let { route, params } = match;
  // 在路由表中查找Controller类
  let Controller = this.routes[route];
  // 如果路由匹配,并且路由表中存在Controller类,则创建一个控制器实例
  if (route && Controller) {
    console.log(match)
    console.log(Controller);
  }

  console.log(url);

  // 如果push参数为true,则添加到history栈中
  if (push) {
    history.pushState({}, null, url);
  }
}
```

执行客户端响应流

将 URL 匹配到控制器中后，接下来就可以执行与服务器端相同的响应流了。

(1) 创建控制器实例。

(2) 执行控制器操作。

(3) 渲染响应。

1. 创建控制器实例

在创建控制器实例时，需要传递一个 context 对象，其中包含了路径参数和查询参数。在服务器端，这些值是从 request 对象中提取出来的，但我们不能在客户端使用这个对象。在下一章中，我们将介绍如何为 request 和 reply 对象创建轻量级的封装，其中包含了路径参数和查询参数的抽象。但目前先来编写 Application 类的 navigate 方法的代码。

```
navigate(url, push=true) {
  // 出于简洁的目的,省略先前的代码

  if (route && Controller) {
    const controller = new Controller({
      query: search,
      params: params
    });
  }

  // 出于简洁的目的,省略后续的代码
}
```

现在我们已经具备在客户端创建控制器实例的能力了，正如我们在服务器端所做的那样。然而，你可能已经发现代码中存在的一个问题了。如果没有的话，请再看一下我们是如何填充 context 对象的 query 属性的。这个值是字符串，而非经过解码的对象，所以我们需要将从 urlParts 解构出来的 search 值解析为一个对象。可能和我一样，你在这些年间已经无数次实现过这个功能，却从没有将这个功能封装起来。好在其他人的代码组织得比我更好，为实现这一目的，我们可以从 npm 安装一个模块。

```
$ npm install query-string --save
```

可以导入这个模块，并使用它来解析 search 字符串。

```
navigate(url, push=true) {
  // 出于简洁的目的,省略先前的代码

  if (route && Controller) {
    const controller = new Controller({
      // 将search字符串解析为对象
      query: query.parse(search),
      params: params
    });
  }
}
```

现在，客户端路由响应实现可以在创建控制器时传递我们希望的参数了，因此控制器就不能分辨出它是在客户端还是在服务器端创建的。非常棒！我们成功地创建了第一个封装！需要再次提醒的是，保持最小化的抽象是非常重要的。在这个示例中，我们之所以在应用框架的代码中进行抽象，只是因为这部分代码不会频繁发生改动，这和应用代码是有区别的。

2. 执行控制器操作

要想在我们创建的控制器实例上执行控制器操作，其方法和服务器端基本相同，如例 8-12 所示。唯一的区别是，需要为 `request` 和 `reply` 参数传递桩函数。我们将在下一章中创建这些封装。

例 8-12　在 `Application` 类的 `navigate` 方法中执行控制器操作

```
navigate(url, push=true) {
  // 出于简洁的目的,省略先前的代码

  if (route && Controller) {
    const controller = new Controller({
      // 将search字符串解析为对象
      query: query.parse(search),
      params: params
    });

    // request和reply的桩函数;下一章将实现其封装
    const request = () => {};
    const reply = () => {};
    // 执行控制器操作
    controller.index(this, request, reply, (err) => {
      if (err) {
        return reply(err);
      }
    });
  }
}
```

3. 渲染控制器响应

在客户端为 ./src/lib/Controller.js 文件中的 `toString` 方法实现另外一种渲染方式。

```
render(target, callback) {
  this.toString(function (err, body) {
    if (err) {
      return callback(err, null);
    }

    document.querySelector(target).innerHTML = body;
    callback(null, body);
  });
}
```

这种方式可以让我们充分利用为客户端优化的各种渲染模式，比如 React.js 的虚拟 DOM（https://facebook.github.io/react/docs/refs-and-the-dom.html），这与使用字符串连接的模板库相对应。

如果在终端运行 Gulp 默认任务，然后在浏览器中打开 http://localhost:8000/hello/{fname}/{lname}，并通过链接进行跳转，你应该能够看到页面变化，但结果并不符合我们的预期。欢迎信息的内容一直是“hello hello.html Sanchez”。这是因为我们还没有针对客户端配置 Nunjucks，也没有在服务器端添加处理器来处理模板文件的请求，现在每次请求都会返回 ./src/index.html 的内容，而且控制器将路径参数 index.html 当成了 `fname`。我们来修复这个问题。在 .src/index.client.js 文件中（如例 8-13 所示）配置 Nunjucks，以便其可以在浏览器端从绝对路径 /templates 中读取内容。

例 8-13　为客户端配置 Nunjucks

```
import Application from './lib';
import HelloController from './hello-controller';
import nunjucks from 'nunjucks';

// 配置Nunjucks以便从dist目录中读取内容
nunjucks.configure('/templates');

const application = new Application({
  '/hello/{name*}': HelloController
}, {
  // target参数设置控制器响应的内容应该插入到哪个元素中
  // 值为元素的查找选择器
  target: 'body'
});

application.start();
```

现在 Nunjucks 会对 /templates/{template_file_name} 这个地址发起 Ajax 请求。接下来需要在 ./src/index.js 文件中向服务器端添加一个处理器，以响应合适的模板内容，如例 8-14 所示。

例 8-14　为模板文件定义路由处理器

```
import Hapi from 'hapi';
import Application from './lib';
import HelloController from './hello-controller';
import nunjucks from 'nunjucks';
import path from 'path';

// 出于简洁的目的,省略中间代码

server.route({
  method: 'GET',
  path: '/templates/{template*}',
  handler: {
    file: (request) => {
```

```
      return path.join('dist', request.params.template);
    }
  }
});

// 出于简洁的目的，省略后续的代码
```

现在，如果回到浏览器中并通过链接进行跳转，应该能够看到名称发生了对应的改变，这是因为我们按照图 8-1 所示的方式渲染了控制器响应。成功了！现在我们已经建立好了同构 JavaScript 应用的基础！然而，还有一些小问题留待处理，我们将在下一节中处理这些问题。

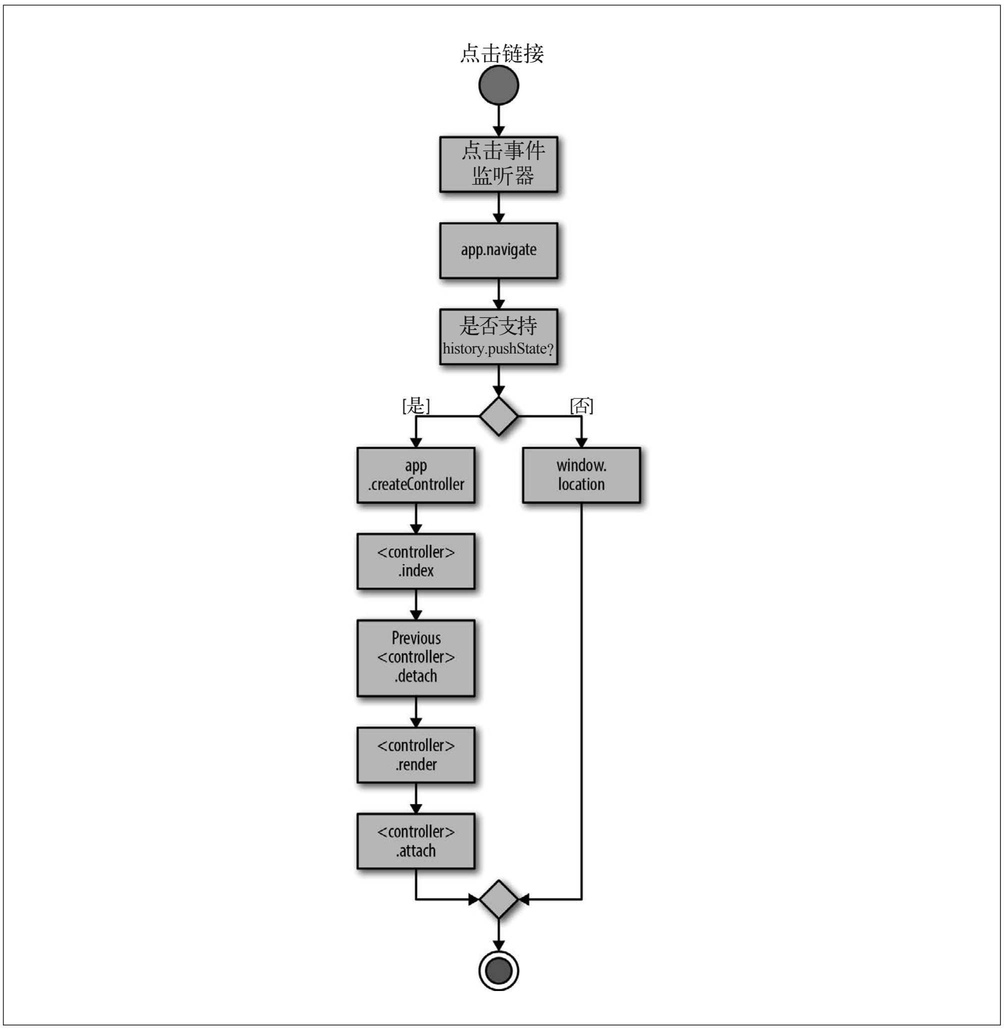


图 8-1：响应 href 点击事件

在这个示例中，我们每次都会为模板进行 Ajax 请求，而且没有缓存内容。这种做法虽然效率不高，但操作简单，延迟了加载，更适用于开发环境。在某些情况下，你可能想要为客户端预编译模板，有时候在服务器端也同样如此。

8.4 组织代码

现在已经完成了本章目标：将框架和应用代码传输到客户端。虽然目前一切正常，但应用中存在一些重复的代码，比如在 ./src/index.js 和 ./src/index.client.js 文件中。这两个文件中都出现了包括路由定义的应用实例化和初始化的代码。这种重复定义路由的方式不太理想，因为当想要添加、删除或者修改路由时，我们必须在两个不同的地方作出修改。此外，我们将应用代码和环境相关的实现细节混在了一起。随着应用规模的增长，这些文件将会变得不同步且难以维护。通过将环境的细节移到不同的选项文件中，可以提高代码的可维护性，以便 ./src/index.js 文件更容易阅读，并成为客户端和服务器端的共同入口点。我们从服务器端的选项文件 ./src/options.js 开始创建，并将特定的环境细节移到这个新的文件中（如例 8-15 所示）。

例 8-15　用于服务器端的应用选项

```
import Hapi from 'hapi';
import path from 'path';
import nunjucks from 'nunjucks';

const server = new Hapi.Server({
  debug: {
    request: ['error']
  }
});
server.connection({
  host: 'localhost',
  port: 8000
});

const APP_FILE_PATH = '/application.js';
server.route({
  method: 'GET',
  path: APP_FILE_PATH,
  handler: (request, reply) => {
    reply.file('dist/build/application.js');
  }
});

server.route({
  method: 'GET',
  path: '/templates/{template*}',
  handler: {
    file: (request) => {
      return path.join('dist', request.params.template);
```

```
    }
  }
});

export default {
  nunjucks: './dist',
  server: server,
  document: function (application, controller, request, reply, body, callback) {
    nunjucks.render('./index.html', {
      body: body,
      application: APP_FILE_PATH
    }, (err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }
};
```

同理，我们还需要创建客户端的选项文件 ./src/options.client.js（如例 8-16 所示）。

例 8-16　用于客户端的应用选项

```
export default {
  target: 'body',
  nunjucks: '/templates'
};
```

现在需要修改 package.json 文件中的属性，以体现这些变化。

```
{
  "browser": {
    "./src/lib/index.js": "./src/lib/index.client.js",
    "./src/options.js": "./src/options.client.js"
  }
}
```

最后，需要修改 ./src/index.js 文件来引入这些新的配置模块，如例 8-17 所示。

例 8-17　统一的应用入口点

```
import Application from './lib';
import HelloController from './HelloController';
import nunjucks from 'nunjucks';
import options from './options';

nunjucks.configure(options.nunjucks);

const application = new Application({
  '/hello/{name*}': HelloController
}, options);

application.start();
```

随着时间的推移和应用规模的增长，这些小变化应该能够帮助我们大大降低应用的开发与维护成本。

8.5 小结

在本章中，我们将框架和应用代码从服务器端转移到了客户端，让其成为了同构代码的基础。我们熟悉了常用的构建模式，在浏览器端沿用了服务器端的生命周期，并利用 History API 来响应 URL 的变化。下一章将在目前的基础之上继续进行构建，为同构应用的一些常用特性创建轻量级的封装。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.4.x"` 命令，可以安装本章的完整代码示例。

第 9 章

创建常用的抽象

Jason Strimpel

在本章中，我们将创建两种经常用于同构 JavaScript 应用的抽象：

(1) 获取和设置 cookie
(2) 重定向请求

通过封装特定环境的实现细节，这些抽象为客户端和服务器端提供了一致的 API。在第二部分中，我们一直在强调抽象带来的危害（包括 Coplien 的观点“抽象是魔鬼”）。尽管如此，本章却是专门介绍如何创建抽象的。让我们先来探讨一下何时应该抽象，以及为什么需要抽象。

9.1　何时抽象，为什么需要抽象

事实上，抽象本身并没有错，但抽象经常被滥用，过早地模糊了重要的实现细节。这些误导性的抽象通常源于人们对编写更加优雅的代码的不懈追求。举例来说，一个用来设置项目脚手架的模块本身是有用的，但如果它对子模块隐藏了一些细节，导致不能被轻易地检查、扩展、配置或者修改，那就另当别论了。正是因为这种滥用被视为魔鬼，进而所有的抽象都受到了牵连而被打上了魔鬼的标签。然而，如果能够适当地利用抽象，那么它将成为一个宝贵的设计工具，可以帮助我们创建直观的界面。

根据我的过往经验，在环境差异的实现细节会给用户带来负担，超过了用户应该关心的范围时，我会使用抽象来标准化跨环境的 API。或者正如 Captain Kirk 所说的那样，我会在

“多数人的需求比少数人（或者某个人）的需求更重要”时进行抽象。不过，这个指导原则并不是万能的，不能仅凭它来决定是否要进行抽象。要想准确知道何时进行抽象非常困难。通常我会思考如下因素。

- 我对这部分代码是否具有足够的领域知识和经验来作出这个决策？
- 我作了太多假设吗？
- 是否存在比抽象更合适的工具？
- 抽象带来的好处是否大于模糊带来的成本？
- 我是否在合适的层次提供了正确的抽象层？
- 抽象是否会隐藏底层对象或函数的内在本质？

如果对这些问题和类似问题的回答能够表明抽象是合适的解决方案，那么你可以继续下去了。不过我还是劝你先和同事讨论一下。我已经数不清有多少次因为忽略了某些关键的信息，使得抽象并没能成为解决问题的最佳方案。

我们已经尽可能地解释了何时以及为什么需要抽象，现在开始创建抽象吧。

9.2　获取和设置cookie

cookie 是纯文本值，创建的最初目的是为了判断两个服务器请求是否来自于同一个浏览器。随后，cookie 还被用于多种用途，其中包括用于客户端的数据存储。浏览器端和服务器端还将 cookie 作为 HTTP 头值传送。这两端都拥有获取和设置 cookie 的能力。因此，对于同构 JavaScript 应用而言，拥有一致的 cookie 读写能力成为了一种普遍的需求，也是我们进行抽象的首选。在读写 cookie 时，进行抽象的难点是客户端和服务器端之间的接口差异很大。此外，在应用层面，即使对环境实现细节非常熟悉，也不能为简化 cookie 的读写提供任何价值。创建一个封装对这些细节进行抽象，并不会减少开发者得到的有用信息，就像 URL（Web 的内部运作机制）对有用信息的抽象不会对用户造成影响一样。

定义API

一个 cookie 由等号分隔的键值对组成，可选属性由分号隔开。

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: bnwq=You can't consume much if you sit still and read books;
Expires=Mon, 20 Apr 2015 16:20:00 GMT
```

在这个示例中，HTTP cookie 作为响应头的一部分被浏览器端接收，也可以作为请求头的一部分发送到服务器端。这种统一的交换格式让客户端和服务器端可以实现接口以获取和设置 cookie。但是，接口在跨环境之间并不一致。这是因为服务器端不像浏览器端那样拥有标准的接口。之所以这样设计，是因为服务器端的职责是多种多样的，这和浏览器端有

很大的差别，后者本身是设计为基于 W3C 标准的一个运行平台，用于展示 UI。正是因为这些差异的存在，我们才需要创建抽象。然而，在创建接口用于跨客户端和服务器端获取和设置 cookie 之前，我们需要先了解不同环境的接口，这也就是说，我们需要具备足够的领域知识来创建适当的抽象。

1. 在客户端获取和设置cookie

在浏览器端获取和设置 cookie 的接口是 `document.cookie`（https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie）。你可以使用 `console.log(document.cookie)` 来打印当前 URL 可访问的所有 cookie。`document.cookie` 返回的键值对以分号隔开。

```
Story=With Folded Hands;Novel=The Humanoids
```

这个字符串值本身比较难用，但我们可以轻松地将其转换成对象，并将 cookie 的名字作为对象的键名，或者我们也可以实现一个函数来根据名字获取 cookie 的值，如下所示。

```
function getCookieByName(name) {
  let cookies = document.cookie.split(';')

  for (let i = 0; i < cookies.length; i++) {
    let [key, value] = cookies[i].split('=');
    if (key === name) {
      return value;
    }
  }
}
```

设置 cookie 的接口是相同的，只是 `document.cookie` 的右侧要赋予新 cookie 的值。

```
document.cookie="bnwq=A love of nature keeps no factories busy;path=/"
```

2. 在服务器端获取和设置cookie

正如 9.2 节中提到的一样，服务器端获取和设置 cookie 的实现方式不尽相同。在 Node 中，可以通过 `http` 模块从请求头中获取 cookie，如例 9-1 所示。

例 9-1　在 Node 服务器端通过名字获取 cookie 的值

```
import http from 'http';

function getCookieByName(name, cookies) {
  for (let i = 0; i < cookies.length; i++) {
    let [key, value] = cookies[i].split('=');
    if (key === name) {
      return value;
    }
  }
}

http.createServer(function (request, response) {
  let someCookie = getCookieByName('some-cookie', request.headers.cookies);
```

```
  response.end('Hello World\n');
}).listen(8080);
```

本书一直使用 hapi 作为应用服务器。hapi 提供了一个更加便捷的接口，如例 9-2 所示。

例 9-2　在服务器端使用 hapi 通过名字获取 cookie 的值

```
import Hapi from 'hapi';

const server = new Hapi.Server({
  debug: {
    request: ['error']
  }

  server.route({
    method: 'GET',
    path: {anything*},
    handler: (request, reply) => {
      let someCookie = request.state['some-cookie'];
      reply('Hello World\n');
    }
  });
});

server.start();
```

在 Node 中，使用 `http` 模块来设置 cookie 是相当简单的（如例 9-3 所示）。

例 9-3　在 Node 服务器端设置 cookie

```
import http from 'http';

http.createServer(function (request, response) {
   response.writeHead(200, {
     'Set-Cookie': 'some-cookie=some value',
     'Content-Type': 'text/plain'
   });
  response.end('Hello World\n');
}).listen(8080);
```

使用 hapi 来设置 cookie 也是很简单的（如例 9-4 所示）。

例 9-4　在服务器端使用 hapi 来设置 cookie

```
import Hapi from 'hapi';

const server = new Hapi.Server({
  debug: {
    request: ['error']
  }

  server.route({
    method: 'GET',
    path: {anything*},
    handler: (request, reply) => {
```

```
      reply('Hello World\n').state('some-cookie', 'some value');
    }
  });
});

server.start();
```

3. 创建接口

我们对 cookie 是如何在客户端和服务器端工作的已经有了更深入的理解，现在是时候创建标准的接口来获取和设置 cookie 了。这个接口（如例 9-5 所示）将取代我们之前编写的特定环境的实现。

例 9-5　同构 cookie 接口

```
class Cookie {

  // name (String): cookie名称
  // value (String): cookie值
  // options.secure (Boolean): 设置https only
  // options.expires (Number): 以毫秒为单位的过期时间
  // options.path (String): 将cookie限制在某个特定路径
  // options.domain (String): 将cookie限制在某个特定域名
  // Returns: Undefined
  set(name, value, options = {}) {

  }

  // name (String): cookie名称
  // Returns: cookie值(String); 默认值为null
  get(name) {

  }

}
```

例 9-5 中描述的这个接口需要提供给路由处理器实例进行调用，以便应用的开发者可以在请求 / 响应生命周期中以及在控制器绑定到客户端之后调用这个 API。要想满足这些要求，其中一种可选方案是在控制器构造函数中创建一个 `context` 对象。

```
constructor(context) {
  this.context = context;
}
```

4. 实现客户端接口

定义好接口后就可以编写客户端的实现了。在 9.2 节的“在客户端获取和设置 cookie”中，我们简单地介绍了一些 cookie 方法，以帮助说明原生浏览器 API 的工作原理。在实际的应用中，获取和设置 cookie 还需要一些额外的工作，比如正确地对值进行编码。

编码 cookie

从技术角度上讲，除了分号、逗号和空格之外，cookie 是不需要对其他字符进行编码的。但是你看过的大多数实现可能都会对值和名称进行 URL 编码，特别是在客户端。需要注意的重点是，在客户端和服务器端要一致地进行编码和解码，此外还要注意不能对值进行双重编码。我们将在例 9-6 和例 9-7 中解决这些问题。

好在很多现成的库已经悄悄地帮我们处理好了这些细节。我们的应用将选用 cookies-js（https://www.npmjs.com/package/cookies-js）这个库，可以通过以下命令对其进行安装。

```
$ npm install cookies-js --save
```

现在使用 cookies-js 编写例 9-5 中定义好的接口。客户端的 cookie 实现（./lib/cookie.client.js）如例 9-6 所示。

例 9-6　客户端的 cookie 实现，保存在 ./lib/cookie.client.js 文件中

```
import cookie from 'cookies-js';

export default {

  get(name) {
    return cookie.get(name) || undefined;
  },

  set(name, value, options = {}) {
    // 将毫秒数转换为秒数,以适配cookies-js的API
    if (options.expires) {
      options.expires /= 1000;
    }
    cookie.set(name, value, options);
  }

}
```

5. 实现服务器端接口

服务器端的实现（./lib/cookie.js）如例 9-7 所示，我们需要简单地包装一下 hapi 的 `state` 接口。

例 9-7　服务器端的 cookie 实现

```
export default function (request, reply) {

  // 编码函数依据rfc文档http://www.rfc-editor.org/rfc/rfc6265.txt
  // 遵循了“cookies-js”客户端实现的同一种模式
  function cleanName(name) {
    name = name.replace(/[^#$&+\^`|]/g, encodeURIComponent);
    return name.replace(/\(/g, '%28').replace(/\)/g, '%29');
  }
```

```
  function cleanValue(value) {
    return (value + '').replace(/[^!#$&-+\--:<-\[\]-~]/g, encodeURIComponent);
  }

  return {

    get(name) {
      return request.state[name] && decodeURIComponent(request.state[name]) ||
        undefined;
    },

    set(name, value, options = {}) {
      reply.state(cleanName(name), cleanValue(value), {
        // 如果值不存在,则使用hapi的默认值
        isSecure: options.secure || false,
        path: options.path || null,
        ttl: options.expires || null,
        domain: options.domain || null
      });
    }

  };

}
```

6. 引入cookie实现

现在实现已经定义好了，是时候将其引入到请求生命周期中了（如例 9-8 和例 9-9 所示）。

例 9-8　在 ./lib/index.client.js 文件中引入客户端 cookie 实现

```
// 出于简洁的目的,省略部分代码
import cookie from './cookie.client';
// 出于简洁的目的,省略部分代码

export default class Application {
  // 出于简洁的目的,省略部分代码
  navigate(url, push=true) {
    // 出于简洁的目的,省略部分代码
    const controller = new Controller({
      // 出于简洁的目的,省略部分代码
      query: query.parse(search),
      params: params,
      cookie: cookie
    });
    // 出于简洁的目的,省略部分代码
  }
  // 出于简洁的目的,省略部分代码
}
```

例 9-9　在 ./lib/index.js 文件中引入服务器端 cookie 实现

```
import cookieFactory from './cookie';

export default class Application {
```

```
  // 出于简洁的目的,省略部分代码
  addRoute(path, Controller) {
    // 出于简洁的目的,省略部分代码
    const controller = new Controller({
      query: request.query,
      params: request.params,
      cookie: cookieFactory(request, reply)
    });
    // 出于简洁的目的,省略部分代码
  }
}
```

7. 代码示例

现在可以在服务器端和客户端设置和获取 cookie 了，如例 9-10 所示。

例 9-10　在 ./src/HelloController.js 文件的 HelloController 类（如例 7-7 所示）的 index 方法中同构设置 cookie 的示例

```
index(application, request, reply, callback) {
  this.context.cookie.set('random', '_' + (Math.floor(Math.random() * 1000) + 1),
  {path: '/' });
  callback(null);
}
```

图 9-1 展示了同构 cookie 的 getter 与 setter 的工作原理。

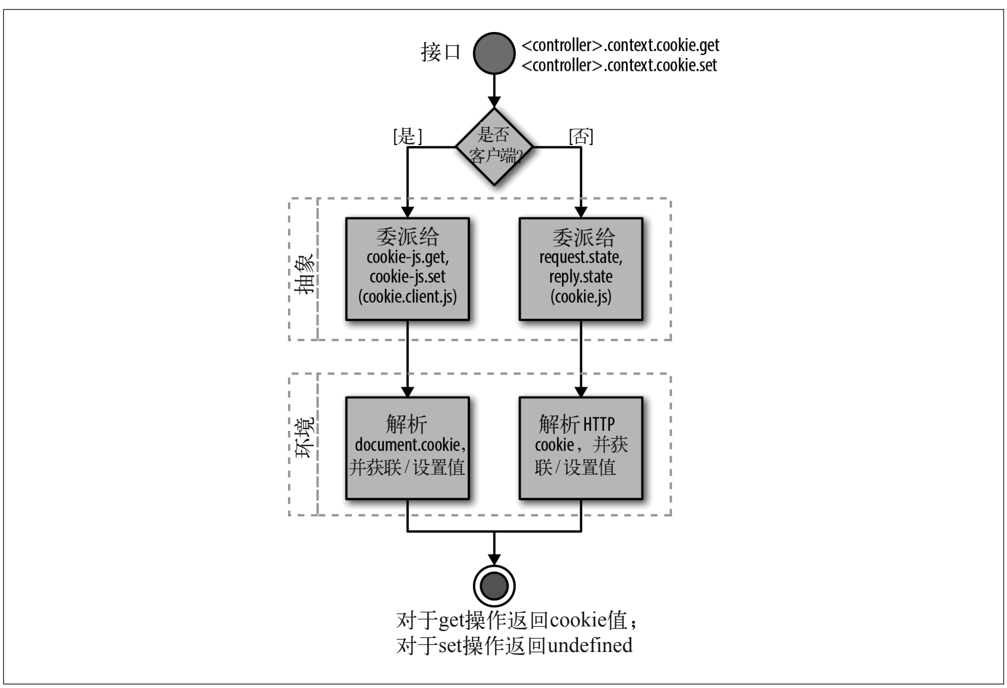


图 9-1：同构 cookie 的 getter 与 setter

9.3 重定向请求

另一个常见的、需要跨客户端和服务器端使用的功能是重定向用户请求。重定向可以让不同的 URL 访问同一个资源。重定向的用例包括个性化 URL、应用重构、认证、管理用户流（如结账）等。在过去，只有服务器端负责重定向请求，通过向客户端回复一个 HTTP 重定向响应的方式进行。

```
HTTP/1.1 301 Moved Permanently
Location: http://theoatmeal.com/
Content-Type: text/html
Content-Length: 174
```

客户端将使用重定向响应中指定的位置来创建新的请求，以确保用户可以接收到最初请求的实际资源。

在重定向响应中，另一个重要的信息是 HTTP 状态码。状态码可以提供给搜索引擎（可以看作另一种类型的客户端）使用，用来确定一个资源是暂时迁移还是永久迁移。如果资源是永久迁移（状态码 301），那么与之前页面相关的所有排名信息都将转移到新的位置中。因此，确保重定向可以在服务器端被正确处理是至关重要的。

定义API

我们在 9.2 节中了解到，虽然在服务器端设置 cookie 的方法各不相同，但是在互联网发送的信息是有标准合约的。在服务器端进行重定向也同样如此。虽然客户端没有创建 HTTP 重定向的概念，但它拥有修改位置（URL）的能力，从而能够发起新的 HTTP 请求。与获取和设置 cookie 一样，修改位置的 API 在跨浏览器之间是一致的。在定义 API 之前，我们再次沿着最佳实践，进一步熟悉将要创建抽象的环境。

1. 在客户端进行重定向

在客户端进行重定向的 API 是 `window.location`（https://developer.mozilla.org/en-US/docs/Web/API/Window/location）。修改 `location` 属性的方法有两种，如例 9-11 所示。

例 9-11　在客户端进行重定向

```
window.location = 'http://theoatmeal.com/';
// 或者
window.location.assign('http://theoatmeal.com/');
```

2. 在服务器端进行重定向

在 Node 环境中，可以利用 `http` 模块进行重定向（如例 9-12 所示）。

例 9-12　在 Node 服务器端进行重定向

```
import http from 'http';

http.createServer(function (request, response) {
  response.writeHead(302, {
    'Location': 'http://theoatmeal.com/',
    'Content-Type': 'text/plain'
  });
  response.end('Hello World\n');
}).listen(8080);
```

在 hapi 中，重定向的写法可以略作简化（如例 9-13 所示）。

例 9-13　使用 hapi 在服务器端进行重定向

```
import Hapi from 'hapi';

const server = new Hapi.Server({
  debug: {
    request: ['error']
  }

  server.route({
    method: 'GET',
    path: {anything*},
    handler: (request, reply) => {
      reply.redirect('http://theoatmeal.com/');
    }
  });
});

server.start();
```

3. 创建接口

在整个请求 / 响应生命周期中，重定向应该都是可用的，以便应用可以在需要时对请求进行重定向。在我们的示例中，这指的就是控制器的操作方法 `index` 被执行时。在服务器端，我们可以直接使用 hapi 的重定向接口 `reply.redirect`。但在客户端，目前只通过 `const reply = () ⇒ {};` 定义了一个没有执行任何操作的 `reply` 函数，因此需要在该函数中添加重定向的功能。有以下两种选择。

(1) 为服务器端的 hapi 的 `reply` 对象创建一个 façade，使其在客户端可以完成相同的功能。
(2) 在客户端的 `reply` 根函数中添加一个重定向的 API，实现与 hapi 相同的功能。

如果选择方案一，那么就可以自由地选择需要创建的 API，但接下来就需要设计接口并创建两种不同的实现。另外，仅仅为了定义我们自己的重定向接口就将整个 hapi 的 `reply` 对象封装起来，这是不是一个好主意呢？对于创建抽象而言，这种程度的需求并不是一个很好的理由。如果选择方案二，那么我们就必须沿用 hapi 的重定向接口，但不需要维护这么多代码，创建的抽象也会少一些。对我们来说，抽象越少越好，特别是在早期，因此我们选择第二种方案。

4. 实现客户端接口

我们将以 hapi 的重定向接口为指导来实现客户端接口。我们只需要实现 `redirect`（https://hapijs.com/api/#response-object-redirect-methods）方法即可，如例 9-14 所示。其他方法不需要进行任何操作，因为这些方法是用来设置 HTTP 状态码的，对客户端来说是无关紧要的。不过，我们仍然需要添加这些方法，以确保在客户端调用其中的某个方法时不会抛出错误。

例 9-14　重定向的客户端实现（./src/lib/reply.js 文件）

```
export default function (application) {

  const reply = function () {};

  reply.redirect = function (url) {
    application.navigate(url);
    return this;
  };

  reply.temporary = function () {
    return this;
  },

  reply.rewritable = function () {
    return this;
  },

  reply.permanent = function () {
    return this;
  }

  return reply;

}
```

5. 引入客户端实现

定义好实现后，现在是时候将其引入到请求生命周期了。我们可以参考例 9-15，将代码添加到 ./lib/index.client.js 文件中。

例 9-15　在 ./lib/index.client.js 文件中引入客户端重定向实现

```
// 出于简洁的目的,省略部分代码
import replyFactory from './reply.client';
// 出于简洁的目的,省略部分代码

export default class Application {
  // 出于简洁的目的,省略部分代码
  navigate(url, push=true) {
    // 出于简洁的目的,省略部分代码
    const request = () => {};
    const reply = replyFactory(this);
    // 出于简洁的目的,省略部分代码
```

```
  }
  // 出于简洁的目的,省略部分代码
}
```

6. 重定向示例

到目前为止，示例中的 `HelloController` 的入口点一直为 `/hello/{name*}`。这很棒，但如果我们想让用户在访问应用根目录 http://localhost:8000/ 时也能看到欢迎信息，该怎么做呢？当然可以设置另一个路由来指向这个控制器，但如果只是想让用户在第一次访问应用时才显示这个欢迎信息，又该怎么做呢？可以利用 cookie 和重定向 API 来处理这个问题（如例 9-16 所示）。

例 9-16 `HomeController` 类的重定向示例（./src/HomeController.js 文件）

```
import Controller from './lib/Controller';

export default class HomeController extends Controller {

  index(application, request, reply, callback) {
    if (!this.context.cookie.get('greeting')) {
      this.context.cookie.set('greeting', '1', {
       expires: 1000 * 60 * 60 * 24 * 365 });
    }

    return reply.redirect('/hello');
  }

  toString(callback) {
    callback(null, 'I am the home page.');
  }

}
```

接下来，需要将这个新的控制器添加到路由中。

```
const application = new Application({
  '/hello/{name*}': HelloController,
  '/': HomeController
}, options);
```

最后，在 hello.html 中添加一个新的链接，以便实现在客户端之间的导航。

```
<p>hello </p>
<ul>
  <li><a href="/hello/mortimer/smith" data-navigate>Mortimer Smith</a></li>
  <li><a href="/hello/bird/person" data-navigate>Bird Person</a></li>
  <li><a href="/hello/revolio/clockberg" data-navigate>Revolio Clockberg</a></li>
  <li><a href="/" data-navigate>Home Redirect</a></li>
</ul>
```

现在，当访问 http://localhost:8000/ 时，客户端和服务器端都可以帮我们重定向到 http://localhost:8000/hello。这种方式让我们可以灵活地实现各种有条件的重定向，并且可以按需

设置相应的 HTTP 返回码。

图 9-2 展示了我们的同构重定向抽象。

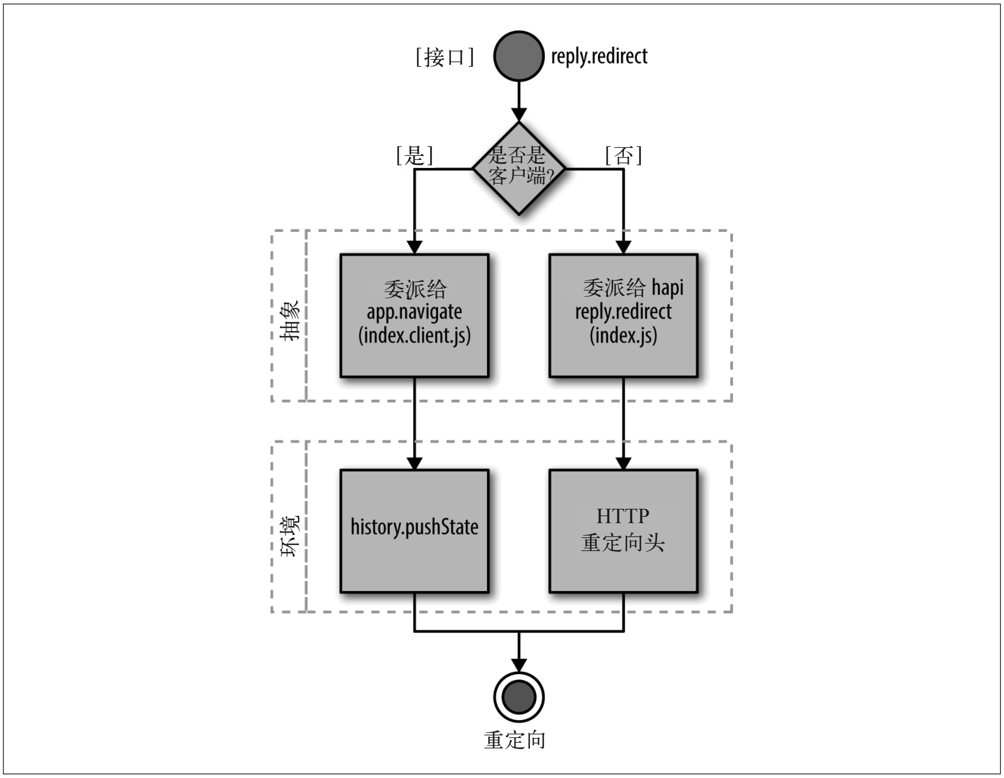


图 9-2：同构重定向

9.4 小结

在本章中，我们创建了两个常见的抽象：获取和设置 cookie 以及重定向，大部分的同构应用都离不开这两个功能。我们还了解了在构建同构 JavaScript 应用的上下文中，应该何时进行抽象以及为什么需要抽象。在未来决定是否应该使用抽象时，这些示例和知识将帮助我们作出更加明智的选择，并在需要抽象时更加合理地使用它。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.5.x"` 命令，可以安装本章的完整代码示例。

第10章

序列化、反序列化和添加事件监听

Jason Strimpel

第 8 章为客户端添加了执行请求 / 响应生命周期的功能。第 9 章又创建了一些同构的抽象，包括获取和设置 cookie 以及重定向用户请求。这些新特性将我们的应用框架从仅适用于服务器端变成了两端通用的解决方案。在实现完整解决方案的路上，我们已经取得了很大的进展，但目前仍然缺少同构 JavaScript 应用的一个关键组成部分：在客户端无缝连接服务器端遗留内容的能力。

从本质上来说，同构应用应该能够获得服务器端渲染的标记并绑定到应用中，就像 SPA 应用在客户端渲染那样。这意味着，在服务器端用于渲染控制器响应的任何数据都应该能够在客户端中使用，以便用户开始与应用进行交互时就可以处理这些数据，如通过表单。为了实现用户交互，DOM 事件处理器同样需要进行绑定。为了在客户端实现 rehydration 过程，必须实现以下 4 个步骤。

(1) 在服务器端序列化数据。
(2) 在客户端创建路由处理控制器实例。
(3) 在客户端反序列化数据。
(4) 在客户端绑定 DOM 事件处理器。

定义 rehydration

在本章的剩余部分和其他同构 JavaScript 的参考资料中，你会频繁地看到术语 rehydration。在同构 JavaScript 应用的上下文中，服务器端使用某种状态来渲染页面响应，rehydration 指的就是重新生成这种状态的行为，其中包括实例化控制器和视图对象，以及基于页面渲染用到的 JSON 数据创建一个或多个对象，比如模型或 POJO（plain old JavaScript object，普通的 JavaScript 对象，https://en.wikipedia.org/wiki/Plain_Old_Java_Object）。基于你的应用架构，rehydration 中还可能包含其他对象的实例化。

本章的余下部分将关注如何在应用中实现这些过程。

10.1 序列化数据

到目前为止，本书示例用到了 cookie、路径与查询参数，以及硬编码的默认值。而在现实世界中，应用还依赖于远程的数据源来获取数据，比如 REST（https://en.wikipedia.org/wiki/Representational_state_transfer）和 GraphQL（https://facebook.github.io/react/blog/2015/ 05/01/graphql-introduction.html）服务。在 JavaScript 应用中，这些数据最终会存储在 POJO 中。数据连同 HTML 和 DOM 事件处理器一起创建出 UI。在服务器端，只能创建出界面的 HTML 部分，因为只有客户端接收到服务器端响应并进行处理后，才能进行 DOM 事件的绑定。客户端的处理、rehydration 以及 DOM 事件的绑定工作通常依赖于服务器端返回的状态数据。此外，rehydration 和 DOM 事件绑定完成后，用户交互会触发这些事件，这时通常需要访问这些数据，以作出修改或者进行判断。因此，在 rehydration 过程及随后的时间中，确保数据的可访问性很重要。问题是，POJO 不能作为 HTTP 请求中的一部分在网络中发送。需要将 POJO 序列化为一个字符串，使其可以传递给模板、被客户端解析以及赋值给一个全局变量。

序列化操作可以通过 `JSON.stringify` 函数（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify）来完成，该方法会创建 POJO 对应的一个字符串表示。虽然这就是序列化 POJO 的标准方法，但我们仍然需要在例 7-3 中的 `Controller` 类的基础上添加一个新的方法，这个新方法可以在服务器端执行，并返回一个序列化后的 POJO。

```
serialize() {
  return JSON.stringify(this.context.data || {});
}
```

对我们的应用来说，默认实现是序列化 ./src/lib/controller.js 文件中的 `this.context.data` 属性。不过，你也可以轻松地覆盖这个默认行为，以应对不同的用例。某些框架或库提供了设置和获取 POJO 数据的 API，比如 Backbone 中的 model（http://backbonejs.org/#Model）

或者 Redux（http://redux.js.org/）中的 store（`store.getState`）。对于上述及其他的客户端数据存储方式而言，`serialize` 函数可以轻松地被重写以实现自定义的行为，并序列化这些对象。

> 正如前面提到的，在 `Controller` 类的 `serialize` 方法中序列化的数据通常是来源于某个远程位置的。这些数据通常使用 HTTP 协议作为传输机制。这在客户端可以通过 Ajax 来完成，而在服务器端则可以通过 Node 的 `http` 模块来实现。获取数据的工作需要特定的客户端实现。好在我们不是第一个需要同构 HTTP 客户端的开发者。很多开源库都可以实现这一功能，其中比较流行的包括 isomorphic-fetch（https://www.npmjs.com/package/isomorphic-fetch）和 superagent（https://www.npmjs.com/package/superagent）。

接下来需要修改例 8-15 中的 `document` 函数（./src/options.js），以调用新的 `serialize` 函数，如例 10-1 所示。

例 10-1　服务器端的应用选项调用了 `Controller` 类的 `serialize` 方法

```
export default {
  nunjucks: './dist',
  server: server,
  document: function (application, controller, request, reply, body, callback) {
    nunjucks.render('./index.html', {
      body: body,
      application: APP_FILE_PATH,
      state: controller.serialize(),
    }, (err, html) => {
      if (err) {
        return callback(err, null);
      }
      callback(null, html);
    });
  }
};
```

最后，需要在模板文件 ./src/index.html 中创建一个全局变量，以便在客户端的 rehydration 过程中进行访问。

```
<script type="text/javascript">
  window.__STATE__ = '{{state}}';
</script>
```

> 确保在引入应用源代码前添加 `<script>` 。

10.2 创建控制器实例

客户端 rehydration 过程中的下一步是为当前的路由创建一个控制器实例。这个实例将会获得序列化后的数据并绑定到 DOM 中，使得用户可以与服务器端返回的 HTML 界面进行交互。好在我们之前已经编写了相关代码，以使用路由器查找相对应的 `Controller` 类，并创建一个控制器实例。只需要对代码进行一下重新组织，就可以重用这些代码了。

在例 8-4 创建的 `navigate` 函数中，我们使用 URL 在应用的路由表中查找 `Controller` 类。如果找到，则创建对应的控制器实例。这部分代码可以移到 `Application` 类（./src/lib/index.client.js）的一个新函数中，使得 `navigate` 可以调用这个新函数，函数的定义如例 10-2 所示。

例 10-2　客户端 `Application` 类的 `createController` 方法

```
createController(url) {
  // 分割路径和搜索字符串
  let urlParts = url.split('?');
  // 解构urlParts数组
  let [path, search] = urlParts;
  // 判断URL路径是否匹配路由器中的路由
  let match = this.router.route('get', path);
  // 解构路由的路径和参数
  let { route, params } = match;
  // 在路由表中查找Controller类
  let Controller = this.routes[route];

  return Controller ?
    new Controller({
      // 将search字符串解析为对象
      query: query.parse(search),
      params: params,
      cookie: cookie
    }) : undefined;
}
```

现在可以在 `navigate` 函数中使用这个 `createController` 函数，以及另一个即将创建的函数 `rehydrate`（如例 10-3 所示）。

例 10-3　客户端 `Application` 类的修改版 `navigate` 方法和 `rehydrate` 方法

```
// 封装的这段代码在rehydrate方法和start方法中的popStateListener都会用到
getUrl() {
  let { pathname, search } = window.location;
  return `${pathname}${search}`;
}

rehydrate() {
  this.controller = this.createController(this.getUrl());
}
```

```
navigate(url, push=true) {
  // 如果浏览器不支持History API,则直接设置location属性并返回
  if (!history.pushState) {
    window.location = url;
    return;
  }

  let previousController = this.controller;
  this.controller = this.createController(url)

  // 出于简洁的目的,省略函数体的余下代码
  // 10.4节中会展示完整的代码示例
}
```

随后 `start` 函数就可以调用 `rehydrate` 函数了（如例 10-4 所示）。

例 10-4　客户端 `Application` 类的 `start` 方法

```
start() {
  // 出于简洁的目的,省略前面的代码
  // 将rehydrate调用添加到函数底部
  this.rehydrate();
}
```

幂等性与 rehydration

在我们的应用中，控制器的构造过程比较简单。然而，在某些情况下，构造过程或者初始化的逻辑可能会更加复杂。在那些场景中，确保初始化逻辑符合**幂等性**（idempotence）是非常重要的。所谓幂等性，就是指无论一个函数执行多少次，其结果都应该是相等的。如果你的初始化过程需要依赖闭包、计数器等，那么这样的初始化逻辑就不是幂等的，导致 rehydration 过程可能会发生错误。举例来说，如果一个应用单例带有状态，但是状态没有传输到客户端，而且没有在 rehydration 过程开始之前将状态添加到单例中的话，那么任何依赖于这份状态的组件初始化逻辑都可能会在客户端发生结果变化。

10.3　反序列化数据

在上一节中，我们在客户端应用的 rehydration 过程中创建了控制器的实例。现在我们需要序列化服务器端创建的状态，并将其作为服务器端页面响应的一部分传输到客户端的控制器实例。这项工作是很有必要的，因为路由可能拥有不同的状态，而应用需要依赖这些状态。例如，如果服务器端渲染的路由具有某些交互性元素或者操作数据的界面，如“添加到购物车”按钮、表单、带有范围选择器的图表等，那么用户很可能会需要用这些数据与页面进行交互、重新渲染页面，或者维持某些变化。

在 10.1 节中，我们将状态数据包含到了页面响应中，因此我们现在要做的就是在基类控制器（./src/lib/controller.js）中实现反序列化方法 `deserialize`。

```
deserialize() {
  this.context.data = JSON.parse(window.__STATE__);
}
```

全局变量

依赖全局变量通常被认为是一种糟糕的做法，因为这会导致潜在的命名冲突以及维护问题。然而，现在并没有什么办法可以完美地解决这个问题。是可以将数据传递到一个函数中，但随后你还是需要通过某种方式将这个函数暴露到全局范围当中。

现在，这个方法可以成为 rehydration 过程的一部分，将数据赋值给控制器实例。我们可以将数据 rehydration 添加到客户端 `Application` 类的 `rehydrate` 方法中（位于 ./src/lib/index.client.js），如下所示。

```
rehydrate() {
  this.controller = this.createController(this.getUrl());
  this.controller.deserialize();
}
```

10.4　添加DOM事件处理器

rehydration 过程的最后一步是将事件处理器添加到 DOM 结构中。拿最简单的方式来说，事件绑定可以通过调用原生方法 `addEventListener` 来完成，如下所示。

```
document.querySelector('body').addEventListener('click', function (e) {
  console.log(e);
}, false);
```

某些库将事件监听函数的上下文设置为创建这个监听器的视图或者控制器对象。Backbone（http://backbonejs.org/#View-events）这样的库提供了接口来定义事件监听器。这些事件监听器通过**事件委托**（https://learn.jquery.com/events/event-delegation/）绑定到包含的视图元素中，以减少浏览器的性能开销。我们只打算在示例中添加一个空方法 `attach` 到控制器中，这个方法会在 rehydration 过程中被调用，其实现细节将由应用开发者来完成。

```
attach(el) {
  // 待应用开发者实现
}
```

除了 rehydration 过程外，`attach` 方法还会作为客户端路由响应生命周期的一部分被调用。这确保了无论是在客户端还是在服务器端，路由处理器都会被绑定到 DOM 中。例 10-5 展示了如何在客户端 `Application` 类的 `rehydrate` 方法中添加 DOM 事件绑定。

例 10-5　在 `rehydrate` 方法中添加 DOM 事件绑定

```
rehydrate() {
  let targetEl = document.querySelector(this.options.target);

  this.controller = this.createController(this.getUrl());
  this.controller.deserialize();
  this.controller.attach(targetEl);
}
```

对 `attach` 方法的两处调用确保了路由监听器总是能够绑定到 DOM 中。然而，这段代码还存在一点问题——只有绑定，没有解绑。如果不断地将事件处理器添加到 DOM 中，而不解绑之前的处理器，那么我们最终可能会面临**内存溢出**（http://www.html5rocks.com/en/tutorials/memory/effectivemanagement/）的问题。为了避免这一问题，可以添加一个 `detach` 方法到基类控制器中，并将其作为客户端路由处理器生命周期的一部分被调用，如例 10-6 所示。与 `attach` 类似，`detach` 也是一个空方法，实现细节留给应用开发者自行编写。

```
detach(el) {
  // 待应用开发者实现
}
```

例 10-6　客户端 `Application` 类的完整版 `navigate` 方法

```
navigate(url, push=true) {
  // 如果浏览器不支持History API,则直接设置location属性并返回
  if (!history.pushState) {
    window.location = url;
    return;
  }

  let previousController = this.controller;
  this.controller = this.createController(url)

  // 如果控制器创建成功,则进行导航
  if (this.controller) {
    // 请求和响应桩函数
    const request = () => {};
    const reply = replyFactory(this);

    if (push) {
      history.pushState({}, null, url);
    }

    // 执行控制器操作
    this.controller.index(this, request, reply, (err) => {
      if (err) {
        return reply(err);
      }

      let targetEl = document.querySelector(this.options.target);
      if (previousController) {
        previousController.detach(targetEl);
      }
```

```
      // 渲染控制器响应
      this.controller.render(this.options.target, (err, response) => {
        if (err) {
          return reply(err);
        }

        reply(response);
        this.controller.attach(targetEl);
      });
    });
  }
}
```

10.5 验证rehydration过程

现在我们已经完成了核心部分，应用可以恢复状态并将事件绑定到 DOM 中，我们应该能够 rehydrate 应用了。要想测试状态恢复是否能够正常工作，可以先在 `HelloController` 类（如例 7-7 所示）的 `index` 方法中生成一个随机数，并将其作为属性保存在 `this.context.data` 中。

```
index(application, request, reply, callback) {
  this.context.cookie.set('random', '_' + (Math.floor(Math.random() * 1000) + 1),
    { path: '/' });
  this.context.data = { random: Math.floor(Math.random() * 1000) + 1 };
  callback(null);
}
```

接下来，需要将 `this.context.data` 添加到 `HelloController` 类的 `toString` 方法中，以作为渲染上下文的一部分。

```
toString(callback) {
  // 这里可以更优雅地使用Object.assign进行处理
  // 但出于简洁的目的,我们没有引入polyfill依赖
  let context = getName(this.context);
  context.data = this.context.data;

  nunjucks.render('hello.html', context, (err, html) => {
    if (err) {
      return callback(err, null);
    }

    callback(null, html);
  });
}
```

现在可以在 hello.html 模板中渲染这个随机数了。

```
<p>hello {{fname}} {{lname}}</p>
<p>Random Number in Context: {{data.random}}</p>
<ul>
```

```
    <li><a href="/hello/mortimer/smith" data-navigate>Mortimer Smith</a></li>
    <li><a href="/hello/bird/person" data-navigate>Bird Person</a></li>
    <li><a href="/hello/revolio/clockberg" data-navigate>Revolio Clockberg</a></li>
    <li><a href="/" data-navigate>Home Redirect</a></li>
  </ul>
```

接下来，在 attach 方法中将 this.context.data 打印出来，并比较值是否一致。

```
attach(el) {
  console.log(this.context.data.random);
}
```

还可以在 attach 方法中添加一个点击事件监听器，以验证传递到方法中的目标元素是否正确，事件监听器会绑定到这个元素中。

```
attach(el) {
  console.log(this.context.data.random);
  el.addEventListener('click', function (e) {
    console.log(e.currentTarget);
  }, false);
}
```

接下来，需要确保能够解绑在 attach 方法中绑定的事件监听器。如果不这样做的话，每次跳转到 /hello/{name*} 路由时都会添加一个新的事件监听器。首先，需要将 attach 方法中的事件监听函数分离到命名函数，以便在移除事件监听时可以传递函数的引用。在 HelloController 类的顶部创建一个函数表达式。

```
function onClick(e) {
  console.log(e.currentTarget);
}
```

现在，可以修改 attach 方法并添加一个 detach 方法。

```
attach(el) {
  console.log(this.context.data.random);
  this.clickHandler = el.addEventListener('click', onClick, false);
}

detach(el) {
  el.removeEventListener('click', onClick, false);
}
```

如果重新加载页面并点击任意地方，你应该可以看到控制台中的 body 元素。如果跳转页面，清空控制台信息并再次点击，你应该只能看到一条日志信息，因为 detach 方法会移除上一个路由中的事件监听器。

图 10-1 高度概括了我们的目标。

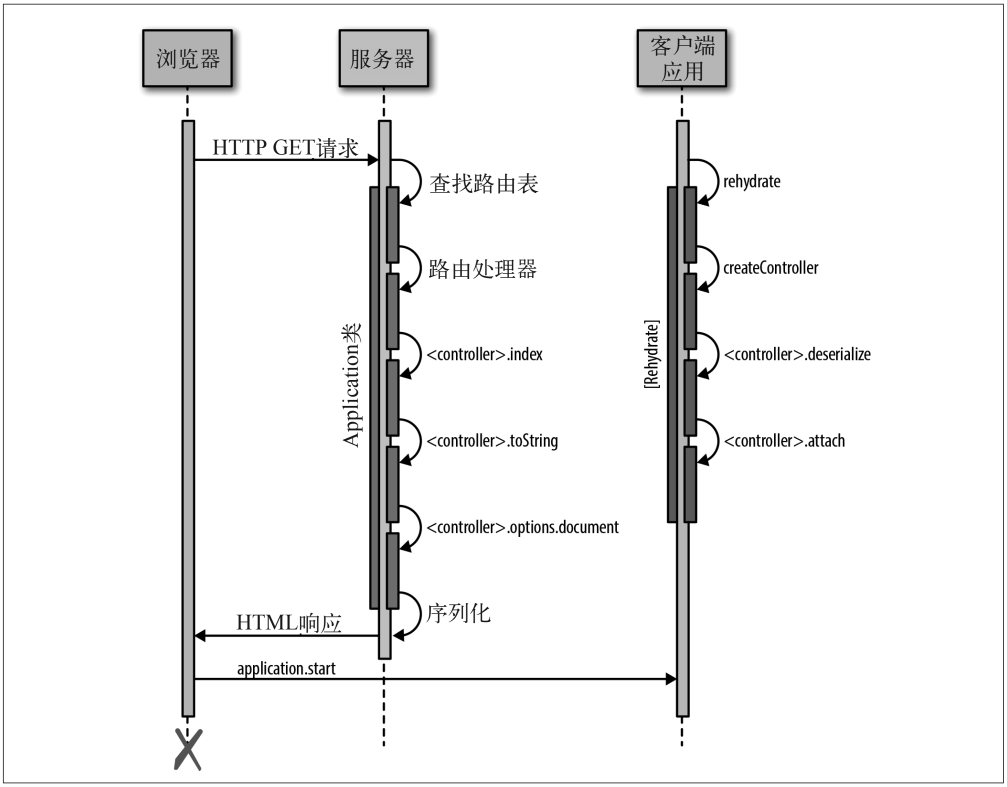


图 10-1：客户端 rehydration

10.6　小结

本章的知识点比较多。我们来快速回顾一下：

- 在服务器端的页面响应中序列化路由数据
- 在客户端创建路由处理器实例
- 反序列化数据
- 将路由处理器绑定到 DOM 中

无论路由是在客户端还是在服务器端进行渲染，以上的 rehydration 步骤都可以确保应用能为用户提供相同的功能，同时也避免了数据的重复请求，优化了性能。

完整的代码示例

通过在终端中运行 `npm install thaumoctopus-mimicus@"0.6.x"` 命令，可以安装本章的完整代码示例。

第11章

结束感言

Jason Strimpel

第二部分涵盖了大量的材料，这些材料可以帮助我们深刻理解同构 JavaScript 应用的工作原理。掌握这些知识后，现在你可以评估和修改现有的解决方案了，或者创造一些全新的内容来满足具体的需求。不过，在勇敢地探索新世界之前，先回顾一下我们介绍过的内容，这有助于更好地准备接下来的同构应用开发工作。我们从快速回顾第二部分开始，包括我们做了什么，为什么要这样做，以及其限制。

11.1　生产准备

在第二部分中，我们逐步建立了一个应用核心，并在此基础之上编写了一个简单的示例。虽然这有助于学习相关概念，但我们不建议你在生产环境中直接使用这个应用核心，因为这个核心是专为本书而编写的，仅作为一种学习工具。对于生产环境而言，还有更好的库可以用于处理一些通用的功能，比如，可以使用 `history`（https://www.npmjs.com/package/history）这个库来管理会话历史。为了专注于学习基本概念和原生 API，我们有意没有使用 `history` 和其他的一些模块，以避免只学习一个库的 API。我们还是建议你利用一些使用量大的、支持度高的开源解决方案，因为这些库能够应对一些边界情况，而我们自己编写的代码可能没有考虑到这些情况。

11.2　衡量架构

在第二部分中，我们强调创建一个通用的请求 / 响应生命周期，实际上你可能并不需要这

样做。例如，如果打算使用 React（https://facebook.github.io/react/）的相关技术栈，那么你可能就不需要这种程度的抽象。互联网上已经有很多文章介绍了使用 React 和其他的开源库来创建一个同构解决方案是多么简单，比如“Exploring Isomorphic JavaScript”（http://nicolashery.com/exploring-isomorphic-javascript/）这篇文章。

注意权衡——Web 和相关的技术一直在发生变化，但公司很少会同意重写整个应用（即使同意了，结果通常也会以失败告终）。如果想适应变化，那么一个轻量级的结构层可以让你轻松地测试新技术，并分阶段地迁移旧代码。当然，你需要在抽象与应用预期的生命周期、用例细节以及其他因素间进行权衡。

> **变化是永恒的**
>
> Web 的快速演变不仅体现在历史中，还体现在依然鼓励你使用代码转译器的如今的 JavaScript 社区中。我们现在编写的代码好像是日新月异的——事实也确实如此。今后 JavaScript 每年都会发布新的版本。请记住，这个新版本的发布计划只是你将面临的改变中的一部分。模式、库和工具的演变更为迅猛。因此，请做好准备迎接这些改变吧！

管理变化

> 无论何处漫游，请快聚首
> 请看清周遭洪水已经围拢
> 面对现实吧，你的骨头将很快被浸透
> 若你的时代仍有价值，值得拯救
> 那就赶快泅
> 否则将沉沦如石头
>
> ——鲍勃·迪伦的歌曲 *The Times They Are a Changin'*

虽然这首歌的创作目的是为了描述一个完全不同的概念，但这部分歌词却很好地描述了 Web 开发者日复一日年复一年面临的斗争：应对变化。新的浏览器 API、新的库、语言的改进、新兴的开发和应用架构模式在不断涌现，这会让你感觉到开发人员的价值和应用的质量将被淹没在这些新的浪潮中。

幸运的是，即使变化的速度越来越快，但同构 JavaScript 应用依然能够维持稳定（至少就其生命周期而言），因此你可以在变化中寻求一点安宁。这是因为同构应用的设计是基于 Web 的 HTTP 请求 / 响应生命周期的。用户使用 URL 对一个资源发起请求，服务器端响应一份净荷数据。唯一的区别是，同构应用的生命周期可以同时在客户端和服务器端使用，因此，在作出架构和实现的决策时，你需要认识到生命周期中的几个关键点。

路由

应该建立一个负责将 URL 映射到函数的路由器。

函数执行

函数应该通过异步的方式执行，即一旦函数执行完成，就应该执行一个回调函数。

响应/渲染

一旦执行完成，就应该通过异步的方式执行渲染。

序列化

在执行、渲染阶段获取和使用过的所有数据都应该成为服务器端响应的一部分。

反序列化

任何对象和数据都需要在客户端重新创建，因为当用户和应用进行交互时，客户端运行时需要用到这些数据。

添加事件

应绑定事件处理器，以便应用可供交互。

就是这么简单。所有其余的补充都是为了增强并实现细节。至于如何将这些步骤连接起来以及利用哪些库，则完全取决于你。例如，你可以选择将一些开源解决方案松散地耦合到一起，又或者仿照本书建立一个更加正式的生命周期，顺带将一些最新的 JavaScript 库融入其中。

> **保持应用核心简洁**
>
> 请记住，向可重用的应用核心中添加的模式与标准越多，其灵活性就越低，并且随着时间的推移，应用在适应变化时会更加容易受损。例如，在准备创建 JavaScript 框架时，我曾经效力的团队决定对 Backbone 和 RequireJS 进行一些标准化的工作，因为这些框架当时还很流行。从那以后，新的模式和库不断出现。这些标准使得应用层很难适应新的变化。这里的技巧就是，要在提供价值和保持灵活性之间找到平衡。

为了支持正式的生命周期所添加的结构的程度应该根据应对变化的需求而定。应对变化的需求也应该和应用的预期寿命保持平衡。如果一个应用将在几年内被取代，那么这个应用可能并不值得我们付出创建一个正式生命周期的代价。而是否需要标准化，即是否打算在此基础上构建多个应用，也应该成为考虑因素。

如果希望在任何时候都具有响应变化的能力，那么在创建应用核心时使用**控制反转**（inversion of control，https://en.wikipedia.org/wiki/Inversion_of_control）来切换库对你来说应该非常重要。最后，不要让其他人来说你需要什么或不需要什么。你比任何人都更加清楚自己的具体情况、同事需求、业务需求以及客户需求。

11.3 小结

这就是第二部分的全部内容。希望你和我们一样享受这段旅程。但本书尚未结束，精彩还在后面。在第三部分中，业界专家大方地贡献了他们的宝贵时间来与我们分享现实世界中的解决方案。我建议你阅读并吸取他们分享的每一份智慧，因为这么做使我大开了眼界，同时对 Web 也有了更加深刻的理解。

第三部分

现实世界的解决方案

现在是同构 JavaScript 的黄金时代。从开始编写本书起，我们见证着同构 JavaScript 世界发生的改变与进化。随着 JavaScript 语言和服务器端 JavaScript 环境的日趋完善，JavaScript 被更广泛地采用。它提供了一个令人难以置信的工具库生态圈，在浏览器端和服务器端发挥着主导作用。

不同背景的开发者正聚焦于 JavaScript。在 2016 年的 O'Reilly Fluent Conference（https://conferences.oreilly.com/fluent/fl-ca）上，我们有幸见到了许多正在使用本书中的概念与实现的开发人员。我们还有幸与 JavaScript 社区的一些思想领袖进行了交流。我们在讨论中发现，在很多情况下，Node.js 进入企业应用的契机其实就是同构 JavaScript。同构 JavaScript 真正改变了很多团队的做法，这些团队在一些情况下习惯于使用不同的服务器端技术栈，但他们现在不得不承认服务器端 JavaScript 确实有很多好处。同构 JavaScript 的意义现在越来越不容忽视，特别是它对初始页面加载性能以及搜索引擎索引和优化的影响。

直到目前，本书一直专注于为建立同构 JavaScript 应用提供必要的基础知识。我们围绕同构 JavaScript 应用介绍了相关概念，并建立了一个种子项目来实现这些概念。在这一部分中，我们将探讨现有的解决方案，研究一下采用了（或即将采用）同构 JavaScript 的各种开发框架。我们还将看到同构项目应用在不同公司、不同项目、不同技术栈中的例子。这些案例研究概述了团队在构建同构 JavaScript 应用时必须解决的各种问题。我们希望这一部分能为你提供一幅立体的画面，阐明在采用同构 JavaScript 时可供参考的解决方案。

第 12 章

沃尔玛实验室的同构React.js方案

Jason Strimpel、Maxime Najim

在沃尔玛实验室工作时，我们正经历着应用架构的蜕变期。现在，我们逐渐开始从温暖、安全的 Java 茧蜕变出迷人的同构 JavaScript 之翼。我们将在本章中分享这个转变过程。

12.1 物种起源

自 2013 年初以来，沃尔玛经历了许多变化。这是因为执行层领导在多年前就着手改革，以便无缝连接沃尔玛客户从线上到线下的体验。这其中的很大一部分工作是对技术和基础设施的投资。这成为了一个演进的过程，某些项目已经在蓬勃发展，而其他项目则已经停止了。导致项目停止的原因是多样的，但我们一直在吸取失败的教训并不断前进。其中和本书最密切相关的新进展是，walmart.com（https://www.walmart.com/）正逐渐减少使用 Java 和 Backbone + Handlebars 的方式来渲染 UI 层。在软件世界中，没有什么东西会真正消逝，除非企业倒闭或者某个特定计划取消。使用其他技术来解决相同的问题只是换了一种（可能会更好的）解决方式而已（如图 12-1 所示）。



图 12-1：在软件世界中，没有什么东西会真正消逝

我们将 walmart.com 面临的问题交由 React.js 和 Node.js 来解决。

12.1.1　问题

通常而言，人们判断问题是否得以解决仅仅只是简单地将其和现有的解决方案进行对比，如交付 walmart.com。使用新开发的、未经证实的技术栈来实现同样的功能是一个比较耗时的工作，本身不会为企业或者用户带来任何真正的价值。这只是将鸡蛋从一个篮子移到另一个篮子而已。

真正的问题不仅仅是交付 walmart.com，而是考虑如何进行交付，以及如何在多重租户和客户之间扩展交付解决方案，即使这个问题定义不能充分地描述问题的真实范围，以及添加一个新技术栈带来的潜在价值。真正的价值取决于新技术和新工艺是否能比之前的解决方案更好地遵循以下的指导原则：

- 吸引并留住工程人才
- 提高开发效率
- 改善代码质量

遵循这些指导原则最终将降低企业的工程成本，并提供更好、更快的用户体验。例如，如果切换到支持跨服务器端和客户端发布的单一语言与运行时，则可以提高开发效率，进而使得企业可以在跨渠道之间快速地交付产品的增强功能，还能减少开发成本。另一个例子是为工程师提供他们热衷于使用的工具和技术，从而吸引和留住人才。这有助于改进技术栈以及开发人员的知识储备，因为社区是有机地围绕着这些技术而发展的。最后，如果你选择的技术既灵活又可以定义渲染生命周期、事件接口和组合模式的话，那么代码的质量也会显著提高，因为这些核心 UI 模式会定义明确的标准。如果不这样做的话，整个公司的工程师可能会多次做出一样的技术选型，却产出不同的代码设计，这会导致可重用性与

整合问题，并增加开发成本。

类似的示例不胜枚举，但如果将解决方案和决定的关注点放在上述列出的指导原则上，就会比仅仅为了赶时髦而追求新技术的潮流要好得多。

12.1.2 解决方案

正如之前所述，沃尔玛使用了 React 和 Node 来解决上述问题。这是因为这些技术既满足了我们的需求，同时也符合吸引和留住人才、提高开发效率以及提高代码质量的目标。除了能够解决这些典型的问题外，同构 JavaScript 解决方案还提供了以下支持：

- SEO 支持
- 分布式渲染
- 单一代码库
- 优化页面加载
- 单一技术栈与渲染生命周期

React 允许我们在多个地方轻松共享封装好的 UI 渲染逻辑（比如组件），这可以大大提高可重用性以及组件的质量，因为多地共用的需求让组件变得更为强大。它还为应用的开发人员提供了构建 Web 和移动端原生 UI 的通用接口。更重要的是，React 还提供了一个很棒的组合模型，社区也已经定义好了组合模式与最佳实践，例如，我们可以遵循容器组件和展示组件的模式。此外，虚拟 DOM 算法降低了重新渲染整个 UI 的成本，与在代码中手动操作 DOM 节点相比，这简化了对 UI 状态的管理。最后，React 社区非常活跃，其中有很多用于构建和维护应用的支持库、模式和学习案例可供参考。

12.2 React模板与模式

在介绍沃尔玛采用的具体做法之前，我们先来概述一个同构 React 应用的通用模板与模式。

设想与补充信息

下列的代码示例假设你已经具备了 React（https://facebook.github.io/react/）、JSX（https://facebook.github.io/react/docs/jsx-in-depth.html）和 ES6（http://es6-features.org/）的基础知识。如果打算使用 React 或创建同构 React 应用，那么网上已经有很多案例和模板项目可以帮助你入门。

12.2.1 在服务器端渲染

常规方法是运行一个 Node 环境的 Web 框架，如 hapi（http://hapijs.com/）或者 Express（http://expressjs.com/），并调用 React 的 `renderToString` 方法。具体的做法可以是这种做法

的变体，最简单的形式如例 12-1 所示。

例 12-1　以字符串形式渲染组件

```
import Hapi from 'hapi';
import React from 'react';
import { renderToString } from 'react-dom/server';
import html from './html';
import Hello from './hello';

class Hello extends React.Component {
  render() {
   return <div>Hello {this.props.text}!</div>;
  }
}

const server = new Hapi.Server({
  debug: {
    request: ['error']
  }
});
server.connection({
  host: 'localhost',
  port: 8000
});

server.route({
  method: 'GET',
  path:'/{42*}',
  handler: (request, reply) => {
   reply(html({
     html: renderToString(<Hello text="World"/>)
   }));
  }
});

server.start((err) => {
  if (err) {
    throw err;
  }

  console.log('Server running at:', server.info.uri);
});
```

来自 `Hello` 组件的 HTML 字符串被渲染到模板，这个模板会作为处理器的响应返回，如例 12-2 所示。

例 12-2　HTML 文档模板

```
export default function (context) {
  return (`
    <html lang="en">
      <head>
        <meta charSet="utf-8" />
```

```
        </head>
        <body>
          <div id="content">${context.html}</div>
          <!--
           this would be a webpack or browserify bundle
           <script src="${context.js}"></script>
           -->
        </body>
      </html>
    `);
  }
```

这种做法适用于静态站点，而静态站点中的客户端组件不需要依赖数据。要想确保数据能够提供给客户端，需要在服务器端渲染时对需要用到的所有数据进行序列化（如例 12-3 所示）。

例 12-3　对数据进行序列化（修改例 12-1 中的 handler）

```
server.route({
  method: 'GET',
  path:'/{42*}',
  handler: (request, reply) => {
   // 数据可能来源于服务、数据库等
   const data = { text: 'World' };
   reply(html({
      data: `window.__DATA__ = ${JSON.stringify(data)};`,
      html: renderToString(<Hello text={data.text} />)
    }));
  }
});
```

数据随后连同 HTML 字符串一起传递到模板中，如例 12-4 所示。

例 12-4　带有数据的 HTML 文档模板

```
export default function (context) {
  return (`
    <html lang="en">
      <head>
        <meta charSet="utf-8" />
        <script>${context.data}</script>
      </head>
      <body>
         <div id="content">${context.html}</div>
         <!--
          this would be a webpack or browserify bundle
          <script src="${context.js}"></script>
          -->
      </body>
    </html>
  `);
}
```

通常而言，需要添加的下一个新功能是可以在服务器端和客户端之间共享的路由系统。这是不可或缺的，因为如果没有将 URL 映射到处理器（在这个示例中指的是组件）的方

式，那么每个 URL 都会返回相同的响应内容。最常见的路由系统是 `react-router`（https://github.com/reactjs/react-router），因为它是为 React 而设计的，且本身也是一个 React 组件。典型的用法是用 `react-router` 中的 `match` 函数将传入的请求映射到路由，如例 12-5 所示。

例 12-5　在服务器端匹配路由

```
import Hapi from 'hapi';
import React from 'react';
import { renderToString } from 'react-dom/server';
import { match, RouterContext, Route } from 'react-router';
import html from './html';

// 出于简洁的目的,省略部分代码

const wrapper = (Component, props) => {
  return () => {
    return <Component {...props} />
  }
}

server.route({
  method: 'GET',
  path:'/{42*}',
  handler: (request, reply) => {
    // 数据可能来源于服务、数据库等
    const data = { text: 'World' };
    const location = request.url.path;
    const routes = (
      <Route path="/" component={wrapper(Hello, data)} />
    );

    match({routes, location}, (error, redirect, props) => {
      if (error) {
        // 渲染500页面
        return;
      }

      if (redirect) {
        return reply.redirect(`${redirect.pathname}${redirect.search}`);
      } else if (props) {
        return reply(html({
          data: `window.__DATA__ = ${JSON.stringify(data)};`,
          html: renderToString(<RouterContext {...props} />)
        }));
      }
    });

    // 渲染404页面
    return;
  }
});
```

这个示例引入了大量的新代码，细分情况如下。

- `wrapper` 包裹函数负责将 `props` 注入到 `react-router` 的路由处理器中。
- `react-router` 的 `match` 函数连同 `path`（`location`）一起判断是否存在匹配的路由。
- `props` 的存在代表匹配到了路由，此时示例中的 `RouterContext`（即 `Hello` 组件包裹的路由控制器组件）会被渲染。

对代码进行模块化

在现实世界中，这段代码会被切分为可重用的多个模块，以便在客户端和服务器端之间共享。为了保持简化的状态，这段代码采用了单个代码块 / 模块的形式。

12.2.2　在客户端恢复

上一节描述了大部分同构 React 应用在服务器端渲染时执行的基本步骤。虽然每个人的实现方式可能略有不同，但在最后的时候，包含了 React 组件树渲染出的标记字符串的 HTML 文档和相关的数据都被发送到了客户端。接下来由客户端接手服务器端剩余的工作。在同构 React 应用中，只需简单地调用 `ReactDOM.render`，将结果渲染到服务器端 `renderToString` 方法注入的 DOM 节点位置中即可（如例 12-6 所示）。

例 12-6　在客户端渲染

```
import React from 'react';
import ReactDOM from 'react-dom';

class Hello extends React.Component {
 render() {
  return <div>Hello {this.props.text}!</div>;
 }
}

const props = window.__DATA__;

ReactDOM.render(<Hello {...props} />, document.getElementById('content'));
```

客户端渲染与服务器端渲染

在给定相同数据的条件下，组件的 `render` 方法在客户端和服务器端的 `return` 值应该是没有区别的。如果存在区别，那么在调用 `ReactDOM.render` 时，DOM 会被重新渲染，这可能会导致性能问题，同时也会降低用户体验。要想避免这个问题，请确保在给定路由的条件下，传递给客户端和服务器端的数据是一致的。

例 12-6 将会重新渲染组件树，`ReactDOM.render` 创建出的虚拟 DOM 和实际的 DOM 中的所有差异都将被修改。此外还将绑定所有的事件监听器并完成其他工作，详情请参见 React

文档（https://facebook.github.io/react/docs/react-component.html）。当使用 `react-router` 时，前面的原理同样适用，如例 12-7 所示。

例 12-7　使用 `react-router` 在客户端渲染

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router'

class Hello extends React.Component {
 render() {
  return <div>Hello {this.props.text}!</div>;
 }
}

const wrapper = (Component, props) => {
  return () => {
    return <Component {...props} />
  }
}
const props = window.__DATA__;
const routes = (
  <Route path="/" component={wrapper(Hello, props)} />
);

ReactDOM.render(<Router history={browserHistory} routes={routes} />,
  document.getElementById('content'));
```

这个实现非常棒，因为路由现在可以在客户端和服务器端之间共享了。最后一步是确保任意的 rehydrated 数据都可以传递到组件中。通常来说，包裹层或者提供层组件负责处理组件树的数据，如 `react-solver`（https://github.com/ericclemmons/react-resolver）、`async-props`（https://github.com/ryanflorence/async-props）和 `Redux`（http://redux.js.org/）等。选用 `wrapper` 的目的和我们在服务器端的做法保持了一致，都是出于简化的考虑。

> **虚拟 DOM 与校验和**
>
> 当 `render` 方法被调用时，React 使用虚拟 DOM（https://facebook.github.io/react/docs/glosary.html）来表示组件渲染树。虚拟 DOM 用于和真实 DOM 进行比较，二者的差异将会被修补。因此，当 `ReactDOM.render` 在客户端被调用时，除了事件监听器需要绑定外，两者不应该有任何差异。React 使用 `data-react-checksum` 属性来实现这项机制，我们将在 12.4.2 节中对此作更详细的介绍。

以上的步骤可以快速构建一个简单的同构 React 应用。如你所见，这些步骤可以轻松地和第二部分中描述的步骤对应起来，其中的一些步骤是合并的或隐含出现的（例如，`ReactDOM.render` 绑定了事件监听器并创建了控制器实例）。

12.3 沃尔玛采用的方法

沃尔玛对 12.2 节中的实现作出了一些修改。主要的区别是，我们没有在首屏页面加载后过渡到 SPA 模式，原因是这属于优化项。对最小可行产品（minimal viable product，MVP）和 walmart.com 的完全迁移而言，这并不是必需的，而且这也不是我们主要目标的一部分（参见 12.1.1 节）。然而，我们还是在客户端执行了序列化和 rehydrate。从更高层次来看，当服务器端匹配到路由时，则会进行以下这些环节。

- 响应路由子集的一个应用会被初始化。
- 应用创建一个 `redux`（http://redux.js.org/）store，触发引导操作，向 store 提供响应请求所需要的数据。
- `react-router`（https://github.com/ReactTraining/react-router）匹配特定的路由。
- `react-dom/server` 模块的 `renderToString` 方法使用了 `redux` store 和其他数据，匹配的路由随后使用这个方法进行渲染。
- 响应连同序列化后的 `redux` store 数据一同发送到客户端。

在客户端中的环节如下。

- 客户端使用服务器端传来的序列化数据对 `redux` store 进行初始化。
- 应用（一个 React 的 `provider/wrapper` 组件）和 `react-router` 调用 `react-dom` 模块的 `render` 方法。

看起来挺简单的，对吧？从更高层次来看或许如此，但在实际中会遇到很多挑战。我们将在下一节中强调其中的一些挑战。

12.4 克服挑战

> 万事开头难。
>
> ——Jason Strimpel

无论你多么有天分或前期准备得多么充分，你还是难免会犯错，那些精心设计和执行计划还是会出现差错与未知情况。关键在于你将如何应对这些挑战。从 Java 和 Backbone + Handlebars 迁移到 React 和 Node 的过程中，沃尔玛遇到了许多挑战，未来也将如此。接下来我们将概述其中一个挑战。

12.4.1 首字节时间

在准备迁移到 React 时，我们很快就发现首字节时间（Time to First Byte，TTFB）比不上现有的应用。服务器端的 CPU 性能分析显示，大部分时间花费在了 `ReactDOMServer` 的

renderToString 代码上面，用于在服务器端渲染初始标记。

> 首字节时间是衡量 Web 应用服务器性能的一种标准方法。正如其名称所示，该指标就是指浏览器接收页面首个字节所花费的时间。较长的首字节时间意味着应用服务器需要花费很长一段时间来处理请求并生成响应。

事实证明，当页面包含多个虚拟 DOM 节点时，React 的服务器端渲染会成为性能瓶颈。在大型的页面中，`ReactDOMServer.renderToString(..)` 方法会独占 CPU，阻塞 Node 的事件循环，并影响传入服务器的其他请求。这是因为每个页面请求都需要渲染整个页面，甚至包括那些给定相同属性就会返回相同标记的细粒度组件。如果每个页面请求都重新渲染相同组件的话，那么就会浪费大量的 CPU 时间。得出的结论是，要想在我们的项目中使用 React，那么就需要对框架层面进行一些基础性的修改，以减少在服务器端重新渲染需要的时间。

12.4.2　组件渲染优化

我们决定为 CPU 时间腾出空间，为此采用了两种很棒的技术：组件缓存化和组件模板化。

1. 组件缓存化

我们直觉上知道，在给定相同属性的情况下，纯组件总是会返回相同的 HTML 标记。与函数式编程中的纯函数概念类似，纯组件就是属性的函数，这意味着在首次渲染过后，我们可以将渲染的结果记忆（缓存）起来，以达到有效提高渲染速度的效果。因此问题变成了，我们能否避免重新渲染传入相同属性的相同组件，以优化 React 在服务器端的渲染时间？

剖析了 React 代码之后，我们发现 React 有一个 `mountComponent` 函数。组件的 HTML 标记就是在这里生成的。我们知道，如果可以通过 `require` 钩子拦截 React 的 `instantiateReactComponent` 模块，那么我们就不需要 fork React，而是可以直接注入缓存优化逻辑了。例 12-8 是注入的缓存优化的一个简化版本。

例 12-8　使用 `require` 钩子在服务器端缓存组件

```
const InstantiateReactComponent = require("react/lib/instantiateReactComponent");

...

const WrappedInstantiateReactComponent = _.wrap(InstantiateReactComponent,
  function (instantiate) {
    const component = instantiate.apply(
      instantiate, [].slice.call(arguments, 1));
    component._instantiateReactComponent = WrappedInstantiateReactComponent;
    component.mountComponent = _.wrap(
      component.mountComponent,
```

```
      function (mount) {
        const cacheKey = config.components[cmpName].keyFn(
          component._currentElement.props);
        const rootID = arguments[1];
        const cachedObj = lruCache.get(cacheKey);
        if (cachedObj) {
          return cachedObj.markup.replace(
            new RegExp('data-reactid="' + cachedObj.rootId, "g"),'data-reactid="' +
              rootID);
        }
        const markup = mount.apply(
          component, [].slice.call(arguments, 1));
        lruCache.set(cacheKey, {
          markup: markup,
          rootId: rootID
        });

    return markup;
    });
  }
  return component;
 });

Module.prototype.require = function (path) {
  const m = require_.apply(this, arguments);
  if (path === "./instantiateReactComponent") {
    return WrappedInstantiateReactComponent;
  }
  return m;
};
```

如你所见，我们使用了一个 LRU 缓存（Least Recently Used，最近最少使用）来存储已渲染组件的标记（适当替代了 `data-reactid` 属性）。由于不仅想要实现某个接口，还需要获得缓存任何纯组件的能力，因此我们创建了一个可配置的组件缓存库，以接收一个包含组件名称的对象，并传递给 `cacheKeyGenerator` 函数，如例 12-9 所示。

例 12-9　可配置的组件缓存库

```
var componentCache = require("@walmart/react-component-cache");

var cacheKeyGenerator = function (props) {
  return props.id + ":" + props.name;
};

var componentCacheRef = componentCache({
  components: {
    'Component1': cacheKeyGenerator,
    'Component2': cacheKeyGenerator
  },
  lruCacheSettings: {
  // LRU缓存配置项,见下文
  }
});
```

通过指定组件名称并引用 `cacheKeyGenerator` 函数，应用的所有者可以配置这个缓存。这个函数返回一个字符串以表示所有传入的组件渲染，并将该字符串用作优化渲染的缓存键。当后续的组件渲染碰到相同的名称时，将会命中缓存并返回缓存的结果。

使用 React 的最初目的就是在不同的页面和应用之间重用组件，所以我们已经拥有了一套可重用的纯组件以及定义明确的接口。在给定相同属性时，这些纯组件总是会返回相同的结果，且该结果不依赖于应用的状态。正因如此，我们可以使用这里所示的可配置组件缓存代码，以便缓存页面的全局页眉、页脚中的大部分组件，且无须修改组件代码本身。

2. 组件模板化

上述的解决方案已经达到了减少部分服务器端 CPU 资源消耗的目的。但我们想要深化这种缓存优化机制，通过组件模板化允许缓存的渲染标记可以包含更多的动态数据。虽说纯组件本应总是渲染相同的标记结构，但某些属性应该比其他属性更加动态化。例 12-10 中这个简单的 React 商品组件就是一个很好的示例。

例 12-10　React 商品组件

```
var React = require('react');

var ProductView = React.createClass({
  render: function() {
    var disabled = this.props.inventory > 0 ? '' : 'disabled';

    return (
      <div className="product">
        <img src={this.props.product.image}/>
        <div className="product-detail">
          <p className="name">{this.props.product.name}</p>
          <p className="description">{this.props.product.description}</p>
          <p className="price">Price: ${this.props.selected.price}</p>
          <button type="button" onClick={this.addToCart} disabled={disabled}>
            {this.props.inventory ? 'Add To Cart' : 'Sold Out'}
          </button>
        </div>
      </div>
    );
  }
});

module.exports = ProductView;
```

这个组件接收的属性包括商品图片、名称、描述以及价格。如果按照前面所述的方式来缓存组件，那么就需要一个足够大的缓存来容纳所有的商品。此外，访问量较少的商品更可能遇到缓存未命中的情况。这就是需要添加组件模板化功能的原因。该功能需要将属性分为两组。

模板属性

这是可以被模板化的属性。例如，在 <link> 标签中，url 和 label 就是模板属性，因为对不同的 url 和 label 值来说，标记的结构不会发生变化。

缓存键属性

这是指会影响渲染后的标记结构的属性。比如，商品的 availabilityStatus 属性会影响结果标记（例如，随显示的价格生成的按钮可能是“添加到购物车”或者“到货提醒”）。

可以在组件缓存库中配置这些属性，但此时不能只提供 cacheKeyGenerator 函数了，你需要传递 templateAttrs 和 keyAttrs（如例 12-11 所示）。

例 12-11　包含 template 和 key 属性的可配置组件缓存库

```
"use strict";
// 可用于模板化的实例组件缓存
var componentCache = require("@walmart/react-component-cache");

var componentCacheRef = componentCache({
    components: {
      "ProductView": {
      templateAttrs: ["product.image", "product.name", "product.description",
       "product.price"],
      keyAttrs: ["product.inventory"]
    },
    "ProductCallToAction": {
      templateAttrs: ["url"],
      keyAttrs: ["availabilityStatus", "isAValidOffer", "maxQuantity",
       "preorder", "preorderInfo.streetDateType", "puresoi",
       "variantTypes", "variantUnselectedExp"
      ]
    }
  }
});
```

注意，ProductView 的模板属性全部都是动态属性，且对每个商品都是不同的。在这个示例中，我们还将属性 product.inventory 作为一个缓存键，因为标记会根据库存的具体数量而变化，以启用“添加到购物车”按钮。

配置好模板属性后，相应的属性会在 React 组件渲染周期中变化为模板分隔符（即 ${ prop_name }）。随后模板被编译、缓存、执行，并交给 React 作为标记的备份。缓存键属性用于缓存模板。对于后续的请求而言，组件的渲染会执行短路逻辑，调用一个已经编译过的模板。例 12-12 展示了一个带有模板属性和模板分隔符的组件缓存库。

例 12-12　支持模板化的组件缓存库

```
component.mountComponent = _.wrap(
  component.mountComponent,
```

```
    function (mount) {
      const cacheKey = ...
      const rootID = arguments[1];
      // 迭代配置后的模板属性
      // 并设置属性的模板分隔符
      templateAttrs.forEach((attrKey) => {
         const _attrKey = attrKey.replace(".", "_");
         templateAttrValues[_attrKey] = _.get(curEl.props, attrKey);
         _.set(curEl.props, attrKey, "${" + _attrKey + "}");
      });
      const cachedObj = lruCache.get(cacheKey);
      if (cachedObj) {
        const cacheMarkup = restorePropsAndProcessTemplate(
           cachedObj.compiled,
           templateAttrs,
           templateAttrValues,
        curEl.props);
        return cacheMarkup.replace(
          new RegExp('data-reactid="' + cachedObj.rootId, "g"),
                     'data-reactid="' + rootID);
      }
      const markup = mount.apply(component, [].slice.call(arguments, 1));
      const compiledMarkup = _.template(markup);
      self.lruCache.set(cacheKey, {
        compiled: compiledMarkup,
        rootId: rootID
      });
      return restorePropsAndProcessTemplate(
         compiledMarkup,
         templateAttrs,
         templateAttrValues,
         curEl.props);
  });
```

restorePropsAndProcessTemplate(..) 函数接收模板属性，设置属性键值，并用属性值执行模板编译。

```
  const restorePropsAndProcessTemplate = (
    compiled, templateAttrs, templateAttrValues, props
  ) => {
    templateAttrs.forEach((attrKey) => {
      const _attrKey = attrKey.replace(".", "_");
      _.set(props, attrKey, templateAttrValues[_attrKey]);
    });
    return compiled(templateAttrValues);
  };
```

12.4.3 性能提升

通过应用缓存化和模板化的优化方案，可以将请求的平均时间缩短 40%，而且 95% 的请求效率可以提升近 50%。这些优化可以为我们的 Node 服务器释放更多的事件循环，并使得 Node 可以专注于其最擅长的**异步数据请求**。我们取得了以下成果：每次的页面请求占

用更少的 CPU 时间，且 `renderToString(..)` 阻塞的并发请求数减少。如图 12-2 所示，在优化后，服务器端请求的 CPU 概况看起来会好得多。

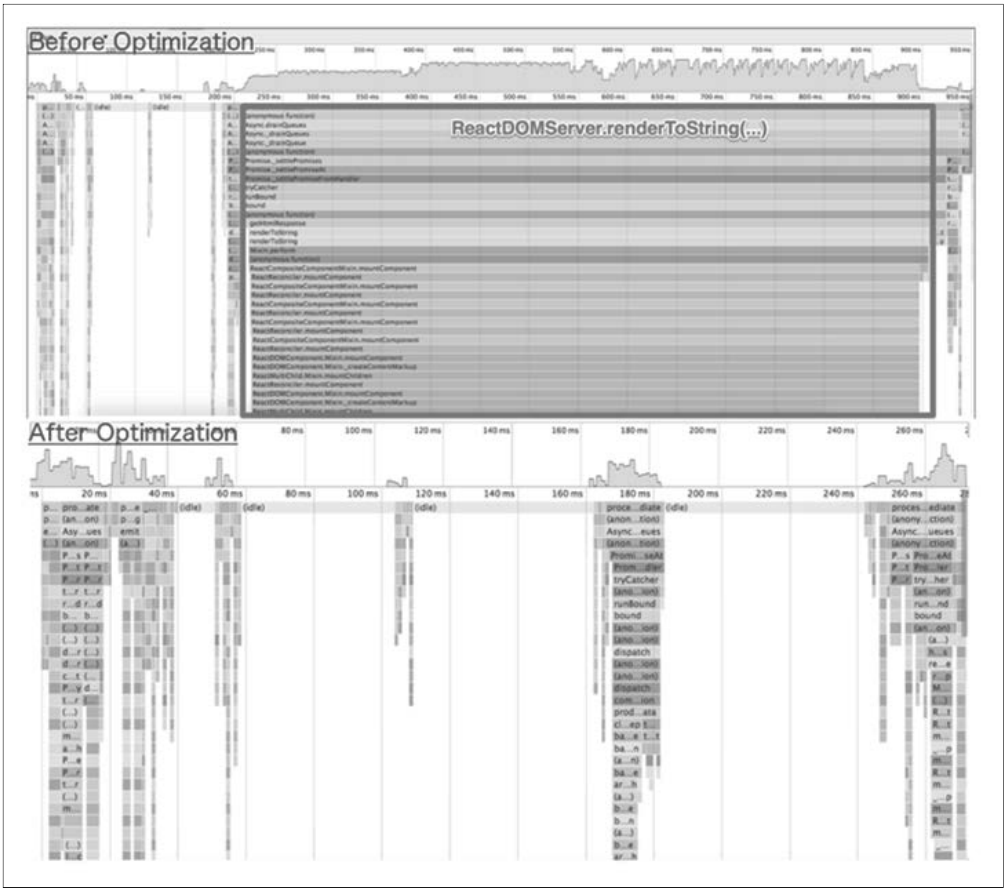


图 12-2：优化前后的 CPU 概况对比

如果高亮显示所有缓存的标记并返回一个示例页面，那么看起来就如图 12-3 所示那样（阴影区域就是在服务器端缓存的标记）。

值得重点说明的是，我们还尝试使用了一些其他的项目来解决 React 的服务器端渲染瓶颈，例如 `react-dom-stream`（https://github.com/aickin/react-dom-stream）和 `react-server`（https://github.com/redfin/react-server）这两个项目就试图通过异步渲染 React 页面来代替同步的 `ReactDOM.renderToString` 方法。替代同步渲染过程后，流式的 React 渲染允许服务器端响应其他的并发请求。流式的初始 HTML 标记还可以让浏览器更早地开始绘制页面（而无须等到整个响应返回）。这些方式有助于提高用户对性能的感知，因为内容可以更快地在屏幕上绘制出来。然而，总体的 CPU 时间不会因此而减少，因为无论采用同

步还是异步的方式，在服务器端需要完成的工作量是不变的。相比之下，组件的缓存化和模板化就可以减少总体的 CPU 时间，因为后续的请求将不再需要重新渲染相同的组件。这些渲染优化可以连同其他的性能优化方案一同使用，其中就包括异步渲染。

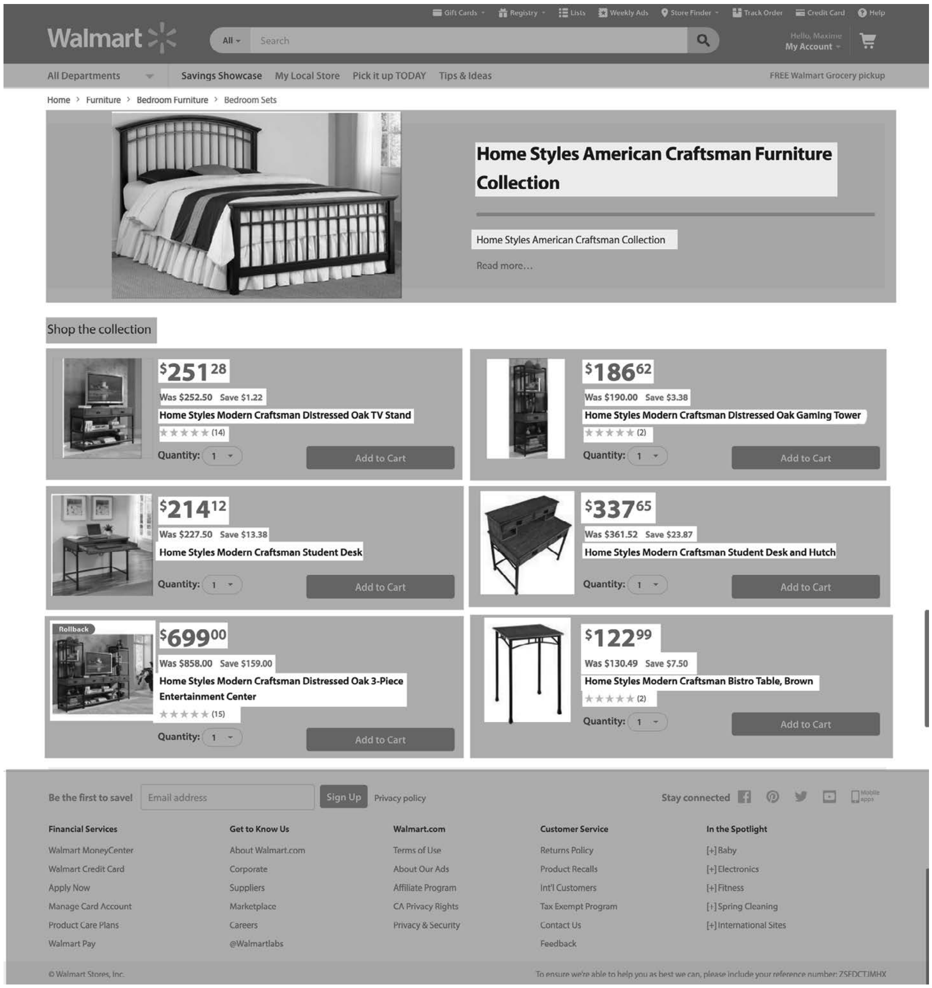


图 12-3：优化后的示例页面（阴影区域表示缓存的标记）

12.5　下一步

接下来将继续在沃尔玛中查找并修复性能瓶颈，其中包括我们在 12.4 节中讨论到的那些问题。准备好解决这些问题后，我们就将正式实现同构并利用 SPA 模型来处理后续的请求。最后，对这一切进行开源！

12.6 感谢

如果经历过类似的转变过程，那么你就会明白这并不容易，特别是对沃尔玛这样的大公司而言。除了技术和规模上的挑战以外，组织、文化以及资源等方面也面临重大挑战。好在团队间的协作、构想、领导力都非常不错。所有成果是公司多个团队共同努力的结果，但我们想要特别感谢沃尔玛实验室的领导对此的投资。我们还要特别感谢 Alex Grigoryan（https://www.linkedin.com/in/alexgrigoryan/）支持我们编写本章的内容，让我们可以分享他负责的产品作出的转变。最后，我们想要感谢 Jack Herrington（https://www.linkedin.com/in/jherr/），感谢他对我们的激励以及他本人的所有开创性工作。我们永远不会忘记他的贡献。

12.7 补充说明

我们很荣幸能够在沃尔玛的变化之旅中贡献出自己的一份力量。至少从我们的角度来看，最好的地方是这个旅程才刚刚开始，这意味着工程师还有很多机会来创建更好的解决方案，以便解决更多难题。因此，如果你也渴望挑战，我们强烈推荐你考虑一下沃尔玛实验室的工作机会。最后，请记得在 Twitter 上关注 @walmartlabs（https://twitter.com/WalmartLabs），留意我们的 GitHub 组织账号（https://github.com/walmartlabs），并持续关注沃尔玛实验室技术博客（https://medium.com/walmartlabs），我们将会向社区分享更多旅途中的细节与代码！

第13章

全栈Angular

Jeff Whelpley

我是在 2012 年 12 月加入的 GetHuman。我的第一项任务非常有趣，需要想办法建立一个丰富的、实时的、流行的旗舰网站 https://gethuman.com/。从本质上来说，我需要寻找方法将两项（在当时）并不常见的要素结合起来。

(1) 一个客户端 JavaScript Web 应用。
(2) 一个 SEO 友好的、性能驱动型的网站。

虽然对上述每个单独的点都有着丰富的经验，但我从未尝试过将它们糅合到一起。在那时候，解决这些需求的方案通常可以归结为类似于 Rails 的实现或是基于无界面浏览器的实现，但这两种方式都有明显的缺点。

Rails 的实现包括建立服务器端的网址，并在不同的页面中加上少量的 JavaScript。虽然这种方式在 SEO 和性能方面表现出色，但无法借助它来利用客户端的一些高级功能。此外，这个解决方案好像更适用于构建传统的网站，这显然并不是我们想要的。我们想要的是快速的、响应式的、流畅的用户体验，通常只能使用客户端驱动的 SPA 才能实现这一目的。

另一方面，一些应用是完全基于客户端构建的，这些应用使用 Backbone、Angular、Ember 或者是纯原生的 JS。要想让客户端应用可以被搜索引擎收录，你需要使用 PhantomJS 这样的无界面浏览器来缓存客户端应用视图的快照。虽然这种做法可以解决问题，但使用无界面浏览器会带来两个大问题。

(1) 运行过慢。
(2) 对于像 GetHuman 这样包含了成百上千个页面的网站来说，这会消耗太多资源。

这两种方式显示都不太适合我们。那么我们该怎么办呢？

13.1 同构JavaScript：Web应用的未来

在尝试了用不同的客户端 / 服务器端方案来满足我们的需求后，我偶然发现了 Spike Brehm 的文章“Isomorphic JavaScript: The Future of Web Apps”（http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/），该文章讲述了作者在 Airbnb 中实现了一种新的 Web 开发方式：同构 JavaScript。Spike 在其中写下了如下内容。

> 归根结底，我们真正想要的是将新旧两种方式结合起来：既要从服务器端生成完整的 HTML 以满足性能和 SEO 需要，又要兼顾客户端应用逻辑的速度与灵活性。为此，我们在 Airbnb 尝试了“同构 JavaScript”应用，这意味着 JavaScript 应用可以同时在客户端与服务器端运行。

这正是我们一直在寻找的答案！ Spike 完美地阐释了我一直以来无法完全归纳出的想法，也是我们在 2013 年花了大半年时间在寻找的东西。

不过还有一个问题。

从概念上来说，我完全认同 Spike 的观点，但他使用的基于 Backbone 的具体解决方案并不是我想要的。虽然对 Backbone 非常熟悉，但我更倾向于使用一个相对较新的框架——Angular.js。因此，为何我不将 Angular 改造为同构的呢？就像 Spike 将 Backbone 改造为同构的那样。

如果对 Angular.js 有所了解，那么你应该知道这说起来容易，做起来难。

13.2 同构Angular 1

为了在 Backbone 上实现服务器端的渲染，Spike 创建了一个完全独立的库，并将其命名为 Rendr（https://github.com/rendrjs/rendr）。他没有对现有的 Backbone 库作出任何修改。我对 Angular 也采取了类似的做法。与 Backbone 类似，Angular 和 DOM 也是紧耦合的。因此，在服务器端实现渲染的方法只有两种，要么对所有的客户端对象（如 `window` 和 `browser`）提供 shim，要么就创建一套高层次的 API，以便应用逻辑可以基于此进行编写。我们选择了前者，这就要求我们基于 Angular 构建一层抽象，类似于 Rendr 与 Backbone 的工作模式。

经过一年多的试验、测试与迭代，我们终于获得了一套优雅的解决方案。

与 Rendr 类似，我们创建了一个名为 Jangular（https://github.com/gethuman/jangular）的库，从而可以在服务器端渲染 Angular 编写的 Web 应用。新版的 GetHuman.com 网站几乎可以在服务器端瞬间渲染出视图，甚至在重负荷情况下也是如此。Angular 客户端在几秒后接管页面，向用户提供几个实时功能。我们认为自己已经解决了服务器端渲染的难题！

但我们的 Angular 1 服务器端渲染解决方案其实仍然存在一些问题。

(1) Jangular 只支持 Angular 功能的一个子集。虽然扩展 Jangular 比较容易，但我们决定刚开始只支持 GetHuman.com 网站需要用到的那部分 Angular 功能。
(2) Jangular 遵循严格的规则，其中包括特定的文件夹层次结构、文件命名规则以及代码结构标准。

换句话说，我们创建的解决方案可以完美地使用于自己的场景，但它很难被其他人所使用。

ng-conf 2015

在 Jangular 被创建出来的几个月之后，我受邀到 ng-conf 进行演讲，介绍我们的 Angular 1 服务器端渲染方案。在会议之前，Google Angular 团队的技术领导 Igor Minar 审核了我的 PPT“Isomorphic Angular”，并告诉我等 Angular 2 诞生后，我尝试做的事情就会变简单很多。那时我并没有十分理解他说的话，但他对我说，他们在 Angular 2 中创建了抽象层，使得 Angular 2 应用无论在客户端、服务器端或其他地方都可以轻松地进行渲染。因此，我在 ng-conf 演讲的最后简单地提及了在 Angular 2 核心中加入服务器端渲染的可能性。当时我是这么说的：

> 我认为，在 Angular 2 中使用这个新功能（不管是由我来实现，还是由很多对此感兴趣的其他开发者来实现），并通过我提供的一些其他功能来创建一个相当酷炫的、大家都可以使用的同构解决方案，这一切只是时间问题。

在演讲结束后，我遇见了 Patrick Stapleton（网名 PatrickJS）。和我一样，他对同构 JavaScript 也很感兴趣。结合我们的想法并经过共同努力之后，Patrick 将 Angular 2 服务器端渲染变成了现实。

这没有花费我们很长时间。在 ng-conf 结束的一周后，Patrick 就取得了一定程度的突破（如图 13-1 所示）。

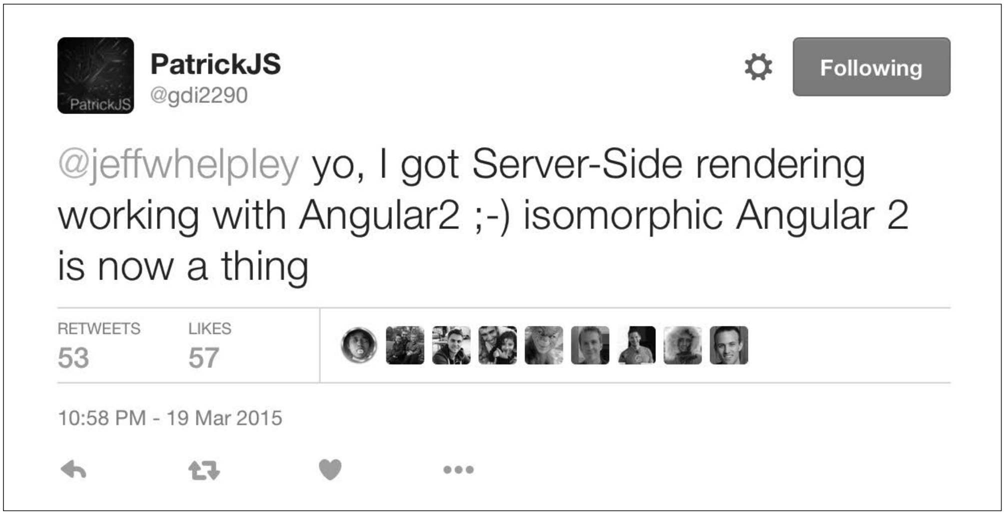


图 13-1：同构 Angular 2！

在和 Angular 社区的开发者及核心团队成员进行了一番讨论之后，Angular 团队的负责人 Brad Green 将我们和 Angular 2 渲染架构的策划者 Tobias Bosch 组织到一起。我们三方开始共同工作。

13.3 Angular 2服务器端渲染

三个月之后，我们建立了一个可用原型，并对 Angular 2 的渲染架构也有了更深入的了解。在 2015 年 6 月，我们在 AngularU 介绍了 Angular 2 服务器端渲染解决方案。

你可以在 YouTube（https://www.youtube.com/watch?v=0wvZ7gakqV4）上观看此次演讲。

下面是本次演讲的总结。

13.3.1 服务器端渲染的用例

下列的每个用例都可以回答这个问题：为什么服务器端渲染对你的客户端 Web 应用很重要？

感知加载时间

通常而言，客户端 Web 应用的初始加载速度都非常慢。Filament Group 近期发布的一份研究（https://www.filamentgroup.com/lab/mv-initial-load-times.html）表明，一个简单的 Angular 1.*x* 应用在移动端设备上的初始页面加载时间平均为 3~4 秒。对更加复杂的应用而言，情况可能更加糟糕。对那些面向客户的应用而言，这是一个最常见的问题，特别是那些经常在移动设备上被访问的应用，不过这也可能是任何应用需要面对的问题。对

于这个用例而言，服务器端渲染的目的是降低初始页面的用户感知加载时间，以便用户可以在一秒内看到真实的内容而不受设备种类或者网络速度的影响。针对这个目标，使用服务器端渲染比使用客户端渲染更有优势。

SEO

虽然 Google 搜索爬虫正不断地改善客户端渲染内容的索引性，SEO 在将来可能不再需要依靠服务器端渲染，但在今天，面向客户的应用仍然需要借助服务器端渲染来提高搜索引擎排行。这是为什么呢？

- 首先，爬虫至今依然还不是非常完善的。在很多情况下，爬虫或许不能准确地索引客户端渲染的内容。这往往是由 JavaScript 的限制或异步加载带来的时序问题所造成的。
- 通过服务器端渲染，爬虫可以准确判断用户需要花费多长时间看到内容（即文档加载完成事件）。要在客户端实现这一点并不容易（而且正如上一个用例中提到的那样，即使能够测量时间，通常也会比服务器端渲染要慢得多）。
- 对于关键字搜索的竞争排名而言，还没有纯客户端 Web 应用击败服务器端渲染网站的任何案例（比如,想想“平板电视”或“2015 年最佳轿车”这样的大件商品的条目）。

浏览器支持

使用更先进的 Web 技术（如 Web Components）的缺点是，这些先进技术难以支持旧版本的浏览器。这就是 Angular 2 不再对低于 IE9 的浏览器提供正式支持的原因。然而，根据应用的具体构建方式，可以将某些富客户端行为放在服务器端进行，以支持旧版的浏览器，应用开发者同时也可以利用 Web 平台的最新特性。以下是两个示例。

- 若应用主要是用于展示信息的，那么可以为使用旧版浏览器的用户提供这些信息，而无须提供任何其他的客户端功能。在这种情况下，可以为使用旧版浏览器的用户提供一个完全由服务器端渲染的网站，而使用新版浏览器的用户将可以获取到完整的客户端应用。
- 若应用必须支持 IE8，则客户端 Web 应用的大部分功能都可以正常工作，只有其中一个组件用到的功能不支持 IE8。对于那一个组件来说，应用可以从服务器端获取完整渲染的这部分 HTML。

链接预览

对于那些提供链接后可以显示网页内容预览的应用，则需要依赖服务器端渲染。由于捕获客户端渲染的 Web 页面比较复杂，这些程序在可预见的未来可能将会继续依赖服务器端渲染。最著名的示例包括 Facebook、G+、LinkedIn 这样的社交媒体平台。与 SEO 的用例类似，这与面向客户的应用的相关性更强。

13.3.2 Web应用脱节

Web 应用脱节（Web App Gap）是我创造的一个术语，表示用户在浏览器中从发起 URL 请求到获得一个带功能的可视页面所需要的时间。对大部分的客户端 Web 应用而言，这段时间花费在等待服务器端响应、下载静态资源、初始化客户端应用框架、抓取数据，并最终将输出绘制到屏幕上。对大多数的 Web 应用而言，这段脱节时间通常需要 3~7 秒，甚至更多。在这段时间里，用户只能看着一个空白屏幕或者一个旋转图标。Filament Group（https://www.filamentgroup.com/lab/mv-initial-load-times.html）的调查显示，在页面加载时间超过 3 秒之后，57% 的用户会放弃打开这个页面。

如今，很多用户已经被移动端的原生用户体验惯坏了。他们不会在乎这是一个网站还是移动应用，他们只想要应用能够立即呈现并提供功能。我们的 Web 应用如何才能做到这一点呢？换句话说，我们要如何消除 Web 应用脱节的这段时间呢？

以下是四种可能的解决方案。

缩短脱节时间

确实存在很多种性能优化的方案，比如缩小客户端 JavaScript 代码或者利用缓存。但问题是，一些因素是你完全无法掌控的，比如网络带宽、客户端设备的性能等。

懒加载

在大部分情况下，客户端的初始请求会导致服务器端返回一份巨大的静荷数据。这份初始静荷数据包含了很多初始视图中不需要用到的内容。如果可以将初始净荷数据减少到只包含初始视图需要的内容，那么它就可能变得很小，加载速度也会快很多。虽然这种方式在理论上是可行的，但却很难实现。我还没发现有哪个库或者框架可以帮助你开箱即用地完成这件事情。

服务工作线程

Addy Osmani 最近提出了使用**应用壳架构**（https://addyosmani.com/blog/application-shell/）的想法，允许在浏览器中运行的服务工作线程下载并缓存所有的资源，以便后续请求某个特定 URL 时应用可以立即从缓存中渲染内容。然而，初始加载可能依然会很慢，因为资源依然需要下载，而且目前并非所有浏览器都支持服务工作线程，具体参见 Can I use...?（http://caniuse.com/#feat=serviceworkers）网站的兼容性列表。

服务器端渲染

初始服务器端响应中包含一个完整渲染的页面，从而可以立即向用户展示页面。随后，在用户开始查看页面并决定做些什么的同时，浏览器会在后台加载客户端应用代码。一旦客户端完成初始化工作并获取到需要的数据，客户端应用就能够接管并控制整个页面的内容。

这个方案已经整合到 Angular 2 中，你可以免费获取。

13.3.3 Angular 2渲染架构

Angular 2 的服务器端渲染架构如图 13-2 所示。

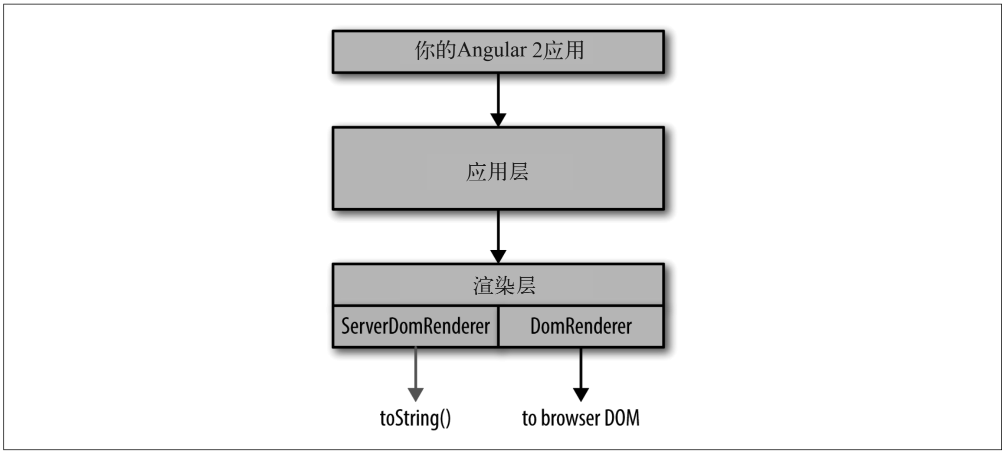


图 13-2：Angular 2 服务器端渲染

我们来分解说明。

Angular 2应用

图的最上方是你基于 Angular 2 编写的自定义代码，这些代码调用了 Angular 2 应用层的 API。

应用层

应用层不需要依赖于任何特定的环境。这意味着，这一层没有直接引用 `window`、`browser` 或者 Node.js 的 `process` 对象。应用层中的所有内容都可以在浏览器端、服务器端或移动设备上运行。这层负责运行你的组件、发起 HTTP 调用，并“编译”你的应用。编译过程的输出结果是一个组件树，其中的每个组件都包含两个主要的数据对象：绑定关系以及某个被称为 ProtoView 的东西，其本质上是一个组件模板的内部表示形式。组件树会从应用层传递到渲染层。

渲染层

渲染层包含两部分。第一部分是可以被应用层引用的一套通用接口。随后接口将交给某个特定的 `Renderer`。默认值是 `DomRenderer`，输出到浏览器 DOM，但这也可以轻松地通过 Angular 2 的依赖注入功能进行改写。而对服务器端渲染而言，我们编写了 `ServerDomRenderer`，将 HTML 以字符串形式输出。服务器端的引导过程将会使用 `ServerDomRenderer` 代替 `DomRenderer`。

唯一需要注意的是，若想让你的 Angular 2 应用在服务器端进行渲染，那你就不能在代码

中直接引用任何的 DOM 对象。例如，你不能使用 `window` 全局对象，而需要使用 Angular 2 API 中的 `Window` 类。这使得 Angular 2 可以限制全局客户端对象的使用，并就在服务器端实现等价的功能提供一定的保证。

13.3.4 Preboot

在演讲接近尾声时，我又抛出了一件我们最近才发现的事情。尽管我们采取的方案可以改善感知性能，但仍然存在一个问题。如果从用户看到服务器端渲染的视图到客户端接管并控制应用的这段时间中，用户与页面发生了交互事件，那么会发生什么事情呢？如上所述，这段时间通常长达 2~6 秒，甚至更长。虽然用户在这段时间里可以看到内容，但如果试图与页面进行交互，结果却是什么事情也没有发生，那么用户就会感到失望（更糟糕的是出现某些意料之外的状况）。对于表单页面，这个问题尤其突出。比如，当用户在这段时间里点击提交按钮，那么会发生什么事情呢？当用户正在文本框中输入内容，而此时视图从服务器端切换到客户端，那么又会发生什么呢？

针对这类情况，以下是一些最常见的解决方案。

(1) 避免使用服务器端渲染客户端的表单。
(2) 在客户端引导逻辑完成执行前，禁用表单元素。

这两种解决方案我们都不喜欢，因而我们创建一个新的库，将其命名为 Preboot，用于在客户端接管前处理服务器视图中的事件。因此，用户就可以在这段时间键入文本框、点击按钮或者在页面上进行任何其他操作，一旦客户端引导执行完成，客户端将无缝处理这些事件。在大部分情况下，页面的用户体验都是能够即时响应功能的，这也正是我们试图实现的。

最棒的是，这个库不是 Angular 2 专用的。你可以将这个库同 Angular 1 一起使用，还可以用在 React、Ember 或者任何其他框架当中，多棒啊！

13.4 Angular Universal

我们的 AngularU 演讲取得了热烈反响，开发者纷纷表示希望能够尽快使用这些功能。Brad 和 Tobias 决定将我们的成果建立为 Angular 的一个官方项目，并称之为 Angular Universal（https://github.com/angular/universal）。

在将代码迁移到 Angular 的官方仓库前，我们所做的大部分工作是完全脱离 Angular 2 的核心库的。在接下来的三个月里，我们开始将其中的一部分放到核心中。在 Angular 2 发布之时，我们希望这个 Angular Universal 库能够尽可能地轻量化，它主要由特定的后端 Node.js 框架的集成库组成，如 Express 或 Hapi.js。

全栈Angular 2

到目前为止，我只谈及了服务器端渲染，这只是全栈 JavaScript 开发这个大概念的其中一部分。服务器端渲染之所以经常备受关注，是因为这可能是最大的技术难点，但我们真正的目的是，无论在何处，都可以使用 JavaScript 来实现所有功能。事实证明，Angular 2 就是为此而生的一个框架。在介绍这一点之前，我们先退一步，解释一下你为什么应该关注全栈 JavaScript 开发。

相信很多人都听过这样一个建议："使用正确的工具做正确的事情"。听起来很聪明，但这只是从某种程度上来说的。问题在于，在实现这个想法时，事情并不一定总会按照你预期的方式来进行。以下是我们面临的一些问题。

委员会

在很多大公司，你并不能直接开始使用某项新技术，而是需要经过好几个委员会的同意（过程总是会很有意思）。当这样做时，最终的决定因素可能就是政治性多于价值本身。

争论

在某些工作环境下，技术的选择可能会导致团队成员因观点不同而关系紧张。

上下文切换

即使不会遇到前两个问题，但需要在一个多元化的技术环境中工作时，你可能就会遇到上下文切换的问题。使用不同的技术工作时，开发者会遇到从一种技术切换到另外一种技术所带来的精神损失；又或许是根据技术界限划分团队带来的额外的沟通成本，造成一些生产力的损失。

代码重复

最后，当拥有一个多元化的技术环境时，重复代码的产生几乎是不可避免的现象。公司通常会遵守一些共同的约定，如安全标准和数据模型等，对于使用的每种语言都要实现一遍这些约定。

你认为有多少人会在工作中遇到上述的这些问题呢？有很多可以解决或者避免这些问题的方法，但我有一个简单的解决方法：只用一把锤子，而将一切都当作钉子。只需要选择一种技术，并用在几乎所有的场合。如果你能做到这一点，那么所有的这些问题都将不复存在。

这在理论上听起来很不错，但事实上，在很长一段时间里，没有一个很好的工具可以成为这个在任何地方都可以使用的魔法之锤。唯一能不受限制运行的语言只有 JavaScript，因此，如果想要尝试使用一种技术来完成所有的事情，你只能选择 JavaScript。

这听起来有点奇怪，因为 JavaScript 在浏览器端的使用情况很糟糕，更不用说在服务器端或移动设备上了。但随着时间的推移，状况也会不断改善。现在我们已经快要到达一个转折点了。

因此，我认为仅仅使用原生的 ES5 JavaScript 是难以实现全栈开发的，但可以使用 ES6、ES7 以及 Angular 2 的一些特性后，你将拥有构建不可思议的全栈应用所需要的一切。以下特性可以让全栈开发变得更加简单。

通用的依赖注入

Angular 1 的依赖注入是比较有效的，但多少存在一些缺陷。Angular 2 的依赖注入简直堪称完美。它不仅可以用于客户端或者服务器端，还可以脱离 Angular 单独使用。这使得构建全栈应用非常简单，因为你可以使用依赖注入来替换任何特定容器的代码（比如那些引用了 `window` 对象且与浏览器紧耦合的代码）。

通用的服务器端渲染

上文已经提及。

通用的事件发送

Angular 2 利用 RxJS observable 来处理事件，其工作方式在客户端与服务器端是相同的。而在 Angular 1 中，事件发送功能约束在客户端的 `$scope` 中，因此服务器端不存在这个功能。

通用的路由

与事件发送类似，在 Angular 2 中，路由功能可以同时在客户端与服务器端工作。

ES6模块

我们的模块经过了格式化，全部支持 ES6。你也可以使用一种格式来编写 JavaScript 代码，并用在所有地方。

ES7装饰器（TypeScript）

在进行大型的全栈开发时，通常有一些横切关注点，如安全、缓存等，这些功能可以通过 TypeScript 中的自定义装饰器来轻松实现。

工具（Webpack、JSPM、Browserify）

所有新的模块打包工具都可以接收一个入口点并遍历依赖树，从而打包生成一整个 JavaScript 文件。这对于全栈开发来说格外有用，因为这意味着你不再需要划分 /server 与 /client 文件夹。取而代之的是将客户端的文件从依赖树中提取出来。

13.5 GetHuman.com

之前提到过，我在 Angular 1 中为 GetHuman 创建了一套服务器端渲染的解决方案。在我编写本章时，这套解决方案已经在生产环境中运行长达 6 个月了。那么，我们是否实现了服务器端渲染与全栈开发的目标呢？

我可以明确回答这个问题：是的。思考以下因素。

性能

在大部分情况下，用户在一秒后就可以看到初始视图。

可伸缩性

在一个只有三核的 Web 服务器上，我们可以轻松地支持 1000 多位用户的并发请求。

生产效率

到目前为止，我们只有两个全职的开发人员，负责十几个不同的 Web 应用。

在不断迭代与改进 Angular 1 基础设施的同时，我们也在设计 Angular 2 版本应用的原型。为此，我在 FullStackAngular2.com（http://fullstackangular2.com/）创建了一个开源的电商应用，希望可以借此找到一种理想的方法来构建全栈的 Angular 2 应用。

13.6 补充说明

要想了解更多关于 Angular Universal 和构建全栈 Angular 2 应用的信息，请点击以下链接：

- Angular Universal 源代码（https://github.com/angular/universal）
- Angular Universal 示例（https://github.com/angular/universal-starter）
- 全栈 Angular 2 示例（http://fullstackangular2.com/）
- 我的博客（https://medium.com/@jeffwhelpley）
- Twitter（@jeffwhelpley，https://twitter.com/jeffwhelpley 与 @gdi2290，https://twitter.com/gdi2290）

第 14 章
Brisket

Wayne Warner

Brisket 是一个基于 Backbone.js 的同构 JavaScript 框架。它是我的团队（彭博社，Consumer Web team）负责开发的。在 2014 年 10 月 2 日，通过彭博社的公司账号在 GitHub 发布，项目遵守 Apache 2 的开源协议。

在开发 Brisket 时，我的团队遵循了以下三个原则：

- 代码自由
- 跨环境一致的 API
- 无须关心其中的过程

在深入了解 Brisket 之前，我们有必要先介绍一下构建 Brisket 的原因。

14.1 问题

和大部分框架一样，Brisket 是因产品需求而生的。在 2013 年年末，我的团队负责重新启动 Bloomberg.com 的观点板块，将其作为新的数字媒体 BloombergView.com 发布。产品团队和设计师对这个新站点提出了下列目标：

- 无限滚动
- 带有灯箱效果的弹出式文章
- 响应式设计（移动端优先）

- 操作响应迅速
- 易于 SEO

我们有 4 位工程师与 3 个月的时间（即 12 周）。

当时，我的团队只有构建传统站点的经验——用服务器端渲染的页面及（在浏览器中的）混合式的客户端代码来处理用户交互。传统网站的一个典型示例是 IGN.com（http://www.ign.com/）。我们在服务器端使用 Ruby on Rails 进行渲染，在客户端使用 jQuery、一部分 Backbone.js 以及原生的 JavaScript 来组合客户端代码。我们只构建传统网站，因为它们可以快速地在服务器端渲染出页面，并提供强大的 SEO 支持。

快速渲染页面对一个数字媒体的成功是至关重要的，因为媒体内容是一个相当灵活的产品，如果能够从其他地方更迅速地获取到相同的内容，那我肯定会转而去那里阅读。高效的 SEO（指页面能够被搜索 ）也是至关重要的，因为搜索引擎仍然是数字媒体产品流量的最大影响因素之一。

当网站需要提供很多客户端功能时，传统网站往往就显得力不从心了。在使用传统网站方式构建新站点时，我们预计会存在以下问题。

无法共享模板与业务逻辑

无限滚动、灯箱效果式的文章这样的功能需要使用客户端渲染。由于我们的服务器端模板是使用 Ruby 编写的，因此不能在客户端重用它们。我们不得不使用 JavaScript 重新创建一套模板。使用两套模板还迫使我们必须维护两套数据模型（一套 Ruby，一套 JavaScript），但实际上它们做的却是同一件事情。

糟糕的功能封装

就传统网站而言，服务器端负责渲染标记，随后客户端代码为其增添功能。如果采用多种语言编写一项功能的代码，并（很可能）存放于文件系统的不同目录中，那么就更加难以说清楚这项功能的完整生命周期。

页面切换的感知速度缓慢

在传统网站中，点击打开一个新页面往往让人觉得比较慢（即使事实上未必如此）。在获取新页面的这个过程中，浏览器与服务器端进行往返来重新渲染页面的所有内容，并重新运行客户端代码，页面之间的过渡感觉上总是比实际要慢。

在 SPA 中，服务器端只渲染最小化标记，客户端应用渲染内容并处理交互。这样看来，SPA 更加合适新站点。从更广泛的角度来看，SPA 就好比潘多拉。SPA 可以将所有的应用代码整合在一起，提供更快的页面切换感知速度并构建丰富的 UI。然而，SPA 并不是万灵药。以下是 SPA 的一些缺点。

初始页面加载速度缓慢

尽管在切换页面时感觉很快，但 SPA 的初始页面的加载速度往往比较慢。这是因为在加载初始页面时需要下载应用的所有静态资源，并启动应用。在应用启动之前，用户看不到任何内容。

缺少SEO

由于 SPA 通常不会在服务器端渲染标记，因此并不能产生任何内容供搜索引擎爬取。在 2013 年年末，搜索引擎才开始尝试爬取 SPA。然而，这项技术还不成熟（具有风险性），我们不能仅依赖于它。

传统站点和 SPA 优势互补，即一方的优势可以解决另一方的短板。于是团队发问了："如何才能从这两种方法中得到所有的优势呢？"

14.2 两全其美

Spike Brehm 在一篇文章（http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/）中提出了 Isomorphic JavaScript 这一术语，这篇文章介绍了 Airbnb 开发的 Rendr 库，阐明了 JavaScript 是可以在客户端与服务器端之间共享的。通过服务器端的 Node.js，Rendr 率先使用了相同的代码库来渲染页面，而在浏览器中则使用 SPA 来处理客户端功能。虽然共享代码库这种方式确实是我们想要的，但这样做可能会产生牺牲现有工具（包括 Ruby、Rails、jQuery 等）的风险。从本质上来讲，团队将重建我们所有的基础设施。但在冒这么大的风险前，我们探讨了一些选择。

我们考虑的第一种方式是，编写一个 SPA 应用并使用无界面浏览器在服务器端渲染页面。但我们最终觉得在服务器端构建两个应用（一个 SPA 和一个无界面浏览器）太复杂了，而且容易出错。

接下来，团队对 2013 年年末可供选择的同构框架进行了探索。然而，我们没有找到很多可靠的选择，那时的大部分同构框架还不太成熟。在探讨过的框架中，Rendr 是为数不多的已在生产环境中运行过的一个。由于看起来风险最低，因此我们决定尝试使用 Rendr。

我们使用 Rendr 构建了一个 BloombergView 的工作原型，但我们希望可以在以下这些领域具有更大的灵活性。

模板

Rendr 默认是附带 Handlebars 模板的，但我们更倾向于使用 Mustache 模板。从 Handlebars 切换到 Mustache 似乎并不是一项简单的任务。

文件组织

与 Rails 类似，Rendr 在某些场景下更倾向于使用约定而非配置。换句话说，应用必须

遵循特定的目录结构才能正常工作。在这个项目中，我们想要通过领域驱动来划分目录结构（即文章、首页、搜索目录），而不是通过功能驱动（即视图、控制器、模型）。在我们看来，将与文章相关的所有内容放在一起更容易进行搜索。

代码风格

我们最关心的问题是，在描述一个页面时，Rendr 是如何控制代码风格的。如例 14-1 中的代码所示，Rendr 控制器描述了如何获取数据，但没有太多的灵活性。当数据仅仅是一个简单对象而非模型或集合时，并没有建立路由的清晰说明。此外，由于控制器没有直接访问视图，视图将不得不承担一部分我们想要在控制器中管理的职责。

例 14-1　2013 年的 Rendr 控制器

```
var _ = require('underscore');

module.export = {
  index: function(params, callback) {
    var spec = {
      collection: {collection: 'Users', params: params}
    };
    this.app.fetch(spec, function(err, result){
      callback(err, result);
    });
  },
  show: function(params, callback) {
    var spec = {
      model: {model: 'User', params: param},
      repos: {collection: 'Repos', params: {user: params.login}}
    };
    this.app.fetch(spec, function(err, result){
      callback(err, result);
    });
  }
}
```

在探索了其他的解决方案之后，我们决定尝试从头开始构建自己的站点。

14.3　早期Brisket

在花费十二周中的两周进行技术选型后，我们决定在主流客户端 JS 框架（包括 Backbone、Angular 或者 Ember）之上进行构建，以节省时间。这三种框架各有优势，但我们选择基于 Backbone 来构建，因为它是最容易迁移到服务器端工作的。

我们在余下的十周里完成了站点的构建工作。然而，由于时间太紧，很难说清框架是怎样开始构建的，应用是如何完成的。

14.4 成为现实

几个月之后，我的团队要负责构建一个新的应用 BloombergPolitics.com 并重构 Bloomberg.com。这两项工作分别要在两个月和三个月之内完成，且中间没有任何间隔。在如此紧张的时间表下，我们将 BloombergView 变成了一个真正的框架——Brisket。以下是我们可以提取出来的一些关键工具。

`Brisket.createServer`

　　该函数返回一个 Express 引擎，你可以在应用中使用这个引擎来运行 Brisket 应用。

`Brisket.RouterBrewery`

　　构造路由器，支持服务器端与客户端的路由功能。

`Brisket.Model`与`Brisket.Collection`

　　标准 Backbone 模型和集合中与环境无关的实现。

`Brisket.View`

　　`Backbone.View` 的修改版本，可以支持一些核心功能，其中包括重新绑定视图、子视图管理、内存管理等。

`Brisket.Templating.TemplateAdapter`

　　继承这个基类，以告诉 Brisket 如何渲染模板。

`Brisket request/response objects`

　　标准化的请求 / 响应对象，提供 cookie、应用层的引用者、响应状态等设置工具。

所有这些工具的提取与重构始终遵循 Brisket 的核心原则——代码自由、跨环境一致的 API、无须关心其中的过程。

14.5 代码自由

我们从 Rendr 学习到的知识是，除非限制开发者可以编写的范围，创建一个包含模型、路由、输出获取、渲染等功能的完整同构框架是相当困难的。虽然知道 Brisket 不能提供常规 Backbone 应用的自由度，但我们已经尽可能去接近了。

Brisket 应用的唯一要求是，必须从路由处理器中返回一个视图或视图的 promise。其他方面都与编写 Backbone 应用没有太大差别。Brisket 负责将你的视图放入页面，而不是交由每个路由处理器来负责。与 Spring 框架中的路由处理器相似，你的路由处理器的职责就是构造一个对象。框架中的其余一些系统则负责渲染这个对象。

可以将路由处理器想象成一个黑盒子（正如下面的数字所表示的）。在请求初始化阶段，

当用户在浏览器中输入一个 URL 时，Express 会处理这个请求，并将请求转发到应用（Backbone 的渲染引擎）。黑盒子的输入是一个请求，输出是一个单独的视图。视图的接收器 `ServerRenderer` 负责以下几项工作。

(1) 序列化视图，并和路由的布局组合起来。
(2) 发送当前服务器端请求中抓取到的所有数据（又名“引导数据”），以及 HTML 静荷数据。
(3) 渲染元标签。
(4) 渲染页面标题。
(5) 设置 HTML 的基本标签，使得 Brisket 的“应用链接”功能可以正常工作，甚至在没有 Brisket 的时候也是如此。

应用链接指的是任何带有相对路径的锚标签。应用链接用于跳转到其他路由。所有其他类型的链接（如绝对路径、完整 URL、mailto 链接等）的功能在任何其他的 Web 页面中都是一致的。通过设置基本标签，你可以确保在禁用 JavaScript 或用户在 JavaScript 完成下载前点击应用链接时，浏览器会以传统的方式跳转到预期的路径——新内容的完整加载页面。通过这种方式，用户总是能够获取到内容。初始页面的请求过程如图 14-1 所示。

一旦序列化后的视图到达浏览器，你的应用就要承担起服务器端剩余的工作了。一种还不错的理解方法是，视图需要重现出在服务器端原本的状态。要想还原视图，Brisket 需要重新运行服务器端已经触发过的路由。不过，这一次的渲染是通过 `ClientRenderer` 进行的。`ClientRenderer` 知道页面中已经存在的哪些视图是没有被完整渲染的。相反，当视图在浏览器中进行初始化时，如果已经在页面中存在，那么 `ClientRenderer` 就会将它们与其自身对应的序列化版本绑定起来。

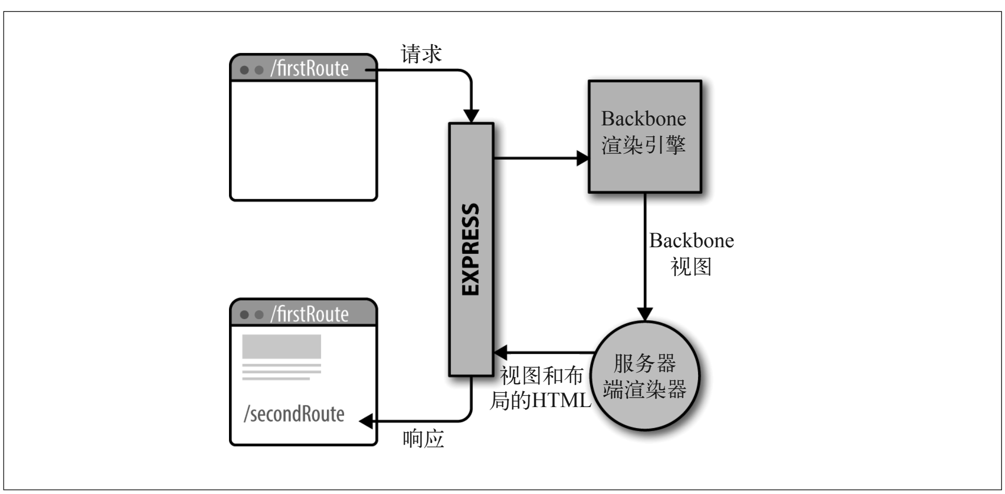


图 14-1：初始页面请求

在完成首次路由后，当用户点击应用链接时，则不会再沿着服务器端获取所有内容，而是应用在浏览器中处理这个请求。路由处理器的工作方式和服务器端相同，即接收一个请求并返回视图。视图会被发送到 `ClientRenderer`，随后将更新布局中的内容。图 14-2 对此进行了描述。

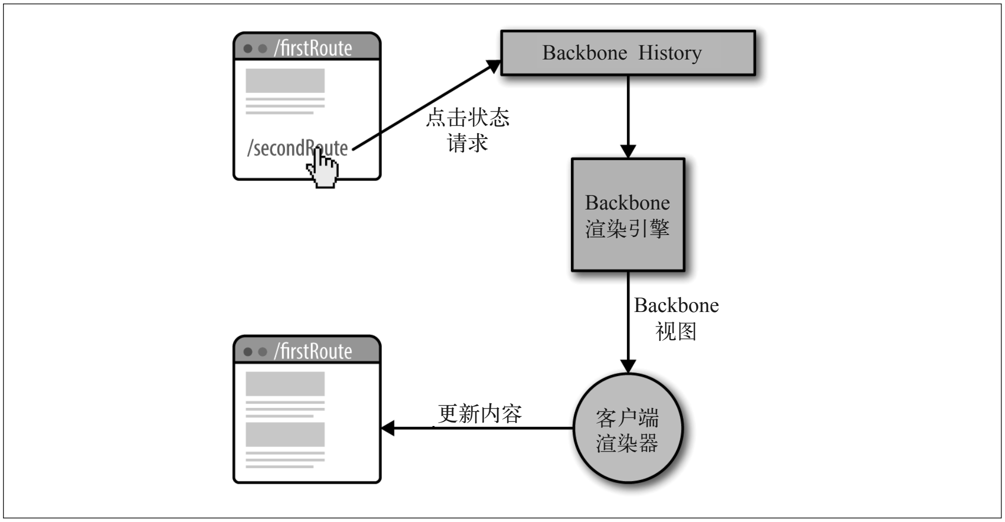


图 14-2：后续的“页面”请求

由于路由处理器接收相同的输入并期望能够产生相同的输出，因此在编写路由处理器时，你无须考虑运行环境这一因素。你可以根据需要来管理路由处理器中的代码。例 14-2 就是 Brisket 路由处理器的一个示例。

例 14-2　Brisket 路由处理器

```
const RouterBrewery = require('path/to/app/RouterBrewery');
const Example = require('path/to/app/Example');
const ExampleView = require('path/to/app/ExampleView');

const ExampleRouter = RouterBrewery.create({
  routes: {
    'examples/:exampleId': 'example'
  },
  example: function(exampleId, layout, request, response) {
    if (!request.cookies.loggedIn) {
      response.redirect('/loginpage');
    }
    request.onComplete(function() {
      layout.doSomething();
    });
    const example = new Example({ exampleId });
    example.fetch()
      .then(() => {
        const exampleView = new ExampleView({ example });
```

```
        exampleView.on('custom:event', console.log);
        return exampleView;
      });
  }
});
```

这个处理器主要负责选择视图来进行渲染，但同时也使用了事件冒泡、基于 cookie 进行判断并在用户可以（“完全”）看到视图时进行某些工作。

你可能还会注意到，构造视图需要用到的对象必须手动选择。Brisket 选择的是基于**配置**的方式，而没有直接约定规则。你可以自由地组织自己的应用，并创建一些对你有帮助的约定。

使用任何模板语言

就我们的新应用而言，我的团队选择使用 Hogan.js（Mustache 模板的编译器）。然而，基于在 Rendr 上进行的尝试，我们想尽量降低切换模板引擎的代价。Brisket 默认提供一个简单高效的 `StringTemplateAdapter`（特别是使用 ES6 模板字符串时），但你可以使用继承在所有视图或某个视图子集中重写这个类。

要想在视图中切换模板引擎，可以继承 `Brisket.Templating.TemplateAdapter`，实现 `templateToHtml` 方法来创建一个自定义的 `TemplateAdapter`。例 14-3 展示了一种实现方式，这种方式使用了一个简单的 Mustache 模板适配器。

例 14-3　简单的 Mustache 模板适配器

```
const MustacheTemplateAdapter = TemplateAdapter.extend({
    templateToHTML(template, data, partials) {
       return Mustache.render(template, data);
    }
});

const MustacheView = Brisket.View.extend({
  templateAdapter: MustacheTemplateAdapter
});
```

改变一个视图的模板引擎只需要几行代码。继承于该视图的任何视图也会使用它的模板引擎。

14.6　跨环境一致的API

通过提供在任何环境中均可预测的一致性 API，Brisket 可以帮助开发者将精力放在应用逻辑本身，而不是“我的代码将在什么环境中运行？”。

14.6.1　模型/集合

Brisket 为 Backbone 的模型与集合创建了与环境无关的实现。从开发者的角度来看，使用 Brisket 创建模型与使用 Backbone 创建模型是完全相同的。在客户端，模型使用 jQuery 来获取数据。而在服务器端，模型同样使用 jQuery 来获取数据。服务器端版本的 jQuery Ajax 数据传输是通过 Node.js 的 `http` 包来完成的。例 14-4 展示了一个简单的 Brisket 模型。

例 14-4　Brisket 模型

```
const Side = Brisket.Model.extend({
    idAttribute: 'type',
    urlRoot: '/api/side',
    parse: function(data) {
       return data.side;
    }
});
```

Brisket 使用 jQuery 来进行客户端传输，因为 Backbone 就是默认使用 jQuery 来传输数据的。为了在两个环境中维护一套一致的 API，Brisket 在服务器端同样使用 jQuery 来获取数据。对于服务器端而言，我的团队更倾向于使用 Node.js 环境的特定工具，如 `http` 或者 `request`。但相比之下，在跨环境中维护一致的 API 显得更为重要。

目前，我们打算抛弃使用 jQuery 传输数据的方式，从而让获取数据变得更加简单、强大。在客户端，我们正研究使用新的 Fetch API。而在服务器端，我们打算切换到 `http` 或 `request`。

14.6.2　视图生命周期

为了让视图保持不受环境的影响，Brisket 劫持原有的 `render` 方法。当 `View.render` 被调用时，Brisket 会执行如下的渲染流程。

(1) 调用视图的 `beforeRender` 回调函数。
(2) 将视图的模型与通过视图的 `logic` 函数指定的任何视图逻辑合并成一个单独的数据对象。
(3) 使用视图的模板适配器、模板，以及第 1 步中得到的数据，将模板渲染到视图的内部元素中。
(4) 调用视图的 `afterRender` 回调函数。

在 Brisket 渲染视图的模板之前，使用 `beforeRender` 回调函数来设置数据、排列子视图、并（或）选择一个模板。`beforeRender` 在客户端和服务器端都会被调用。

在 Brisket 将视图的模板渲染到 `el` 后，使用 `afterRender` 回调函数来修改视图。需要在模板被渲染后完成某些工作（如添加一个特殊的类）时，就可以用上这个函数。`afterRender` 在客户端与服务器端都会被调用。

Brisket 视图还有一个名为 `onDOM` 的回调函数。一旦视图进入到页面的 DOM 结构，则可以使用 `onDOM` 回调函数在客户端对视图作出修改。在 `onDOM` 回调函数中，你可以安全地使用 `window` 对象。所以，如果想要使用一个仅在浏览器中可用的 jQuery 插件，就可以将逻辑放在这个函数中。`onDOM` 只会在客户端被调用。

在视图的渲染流程中，可以确定的是，`beforeRender` 一定会在 `afterRender` 之前被调用。而在客户端，`onDOM` 只会在 `afterRender` 后、视图进入到页面的 DOM 结构中时才会被调用。

除了这些特定的 Brisket 方法外，你还可以沿用处理 Backbone 视图的方式来处理 Brisket 视图。为了促进这一级别的灵活性，我们在服务器端渲染中使用了 `jsdom`。大部分的同构框架都没有使用它，因为对服务器端内容的渲染来说，它相对比较慢和重量级。我们也看到了这一点，但还是决定使用它，因为我们认为其编码的灵活性可以压倒性能影响。我们希望只在需要解析 DOM 时才使用 `jsdom` 替代服务器端的渲染实现。

14.6.3　子视图管理

在标准的 Backbone 应用中，子视图的管理可能会比较棘手。虽然可以在一个视图中直接渲染另一个视图，但要建立视图之间的关系并没有那么简单。另一个痛点是内存管理。Backbone 应用中的一个常见问题是，由于忘记清理用户不再可见的视图而导致内存溢出。在浏览器中，用户只会查看几个页面，内存溢出可能不会导致灾难性的后果，但在同构环境中，客户端的内存溢出意味着服务器端也会产生内存溢出。当内存泄漏严重到一定程度时，这将会导致你的服务器崩溃（我们已经得到了这个教训）。

Brisket 提供了一个子视图管理系统，可以帮助你管理内存并显示子视图。Brisket 的子视图管理着父子视图间的关联信息。当在路由之间进行切换时，它还可以帮助你确保清理视图，而且子视图系统还附带了便捷的方法，可以帮助你将子视图放置到父视图的标记中。子视图系统在所有环境中的工作方式均相同。

14.6.4　跨环境使用的工具

一路走来，我们在多个面向消费者的项目中均使用了 Brisket，遇到的一些问题对过去的传统应用或 SPA 来说比较简单，但在同构应用中却比较棘手。在解决这些问题之后，我们向 Brisket 加入了下列这些功能。

重定向到另一个URL

Brisket 的 `response` 对象提供了一个 `redirect` 方法，该方法的功能和 Express 引擎中的 `response` 对象的功能相同。正如你所期望的那样，该方法可以重定向到你提供的 URL，并提供一个可选的状态码。这在服务器端没有什么问题，但在同构应用中，还有可能会

在浏览器中调用 `response.redirect` 进行路由跳转。那么该如何实现这个方法呢？是使用 pushState 跳转到应用中的另一个路由，还是什么都不做呢？最后，我们决定让客户端在调用 `response.redirect` 时触发新页面的完整加载，就像服务器端重定向所做的那样。此外，通过 `window.location.replace` 方法，确保新的 URL 可以在浏览器历史中取代原本请求的 URL。

请求引用者

在传统的网站中，请求的引用者在服务器端可以通过 Express 中间件的 `request.referrer` 取得，而在客户端则可以通过浏览器的 `document.referrer` 属性取得。在 SPA 中，虽然可以在初始页面加载时通过 `document.referrer` 取得正确的引用者，但随后跳转到新的“页面”时就不能采用这种方式了。我们需要解决这个问题，以便精确地跟踪用户行为，以供分析。标准化的引用者可以通过 Brisket 的 `request.referrer` 属性取得。

14.7 前进之路

Brisket 的灵活性决定了它与第三方工具的互操作性非常高。我们将继续向社区寻求常见问题的优秀解决方案，而且不希望我们的框架会妨碍到这一点。通过尽量简化建立 Brisket 应用时所需要的规则，我们确保有足够的空间来整合任何的第三方代码，哪怕这些代码不是完全用于同构的。

14.7.1 ClientApp与ServerApp

在开始编写 BloombergView 时，我的团队已经对 jQuery 插件非常了解了。在尝试使用了 jQuery 插件之后，我们很快发现，这些插件一般并不是同构友好型的——它们只能在客户端正常地工作。为了能够使用 jQuery 插件和只能用于客户端的其他代码，我们创建了 `ClientApp` 与 `ServerApp` 类。

Brisket 提供了 `ClientApp` 与 `ServerApp` 基类，你可以继承这两个类。你的实现犹如特定环境下的初始化器。你可以在这些类中设置特定环境的日志、初始化 jQuery 插件、根据环境启用或禁用某项功能。

14.7.2 布局模板

Brisket 提供了一个继承于 `Brisket.View` 的 `Layout` 类。在布局模板中，你可以定义页面中的 `<html>`、`<head>` 和 `<body>` 标签。Brisket 允许你为每个路由使用不同的布局。目前，使用多重模板只能在服务器端实现，因为在浏览器中交换布局比较复杂。由于和其他模板一样，布局模板也是标准的标记，因此可以在布局模板中放置通用的第三方嵌入代码，如广告、像素追踪等。

14.7.3 其他经验教训

Brisket 对我的团队帮助很大。我们在非常紧张的期限之内完成了任务（或者只超期了一点）。尽管这些网站取得了成功，但我们在过程之中也吸取了很多教训。以下是其中的一些经验教训。

避免整个打包客户端代码

SPA 的一个普遍问题是，CSS 和 JavaScript 文件庞大。随着应用的发展，代码打包会不断增长，初始页面加载就会变慢。我们正在向 Brisket 之中添加新功能，使它可以轻松地拆分应用打包文件。

避免内存溢出

在编写 SPA 代码时，有时你会忘记自己的代码将在服务器端运行。使用单例或者忘记清除事件绑定都会导致内存溢出。一般来说，关闭资源是一项良好的实践，但许多前端开发者在 SPA 出现前并没有注意到这个问题。在 SPA 中（尤其是在同构应用中）必须遵循这项良好的编码实践。

构建自己的框架可能会让人觉得沮丧

虽然自己构建框架是一件让人兴奋的事，但在时间紧迫的情况下，当 Brisket 由于缺少某个工具而无法实现某项功能时，这也会让人感到沮丧。从积极的方面来看，每一次挫折都会促使 Brisket 带来新的改进（或删减）。

14.8 Brisket的下一步？

目前，在多个面向消费者的产品中，我的团队均在生产环境中使用了 Brisket，其中包括我们的旗舰网站 Bloomberg.com。我们将继续改进 Brisket，以便用户可以拥有更加良好的体验，并让我们的开发工作更加愉快而富有成效。尽管有几个大型网站已经在生产环境中使用了 Brisket，但 Brisket 目前仍然是一个未达到 1.0.0 版本的项目。在编写本章时，我们正积极地筹备 1.0.0 版本的发布工作。我们计划中的新功能 / 改进点包括以下几个方面：

- 更容易将一整个 Brisket 应用代码包拆分为多个小包
- 重构服务器端渲染以提升速度
- 继续简化提炼 API
- 解耦 jQuery
- 前瞻性地支持生成器、异步函数以及模板字符串

尽管 Brisket 目前已经满足了我们的要求，但我们还将继续探究新的技术和实现来构建产品。我们的团队不会局限于某项特定的技术中，即使是 Brisket。不管来源，我们总是想要使用最好的技术来满足产品需求。

14.9 补充说明

Brisket 的构建之旅已经走过了一段相当长的时间，当中充满了跌宕起伏。要想了解更多关于 Brisket 的内容及更新情况，可以查阅我们的 npm（https://www.npmjs.com/package/brisket）页面。此外，你也可以尝试使用我们的项目生成器——Brisket 生成器（https://www.npmjs.com/package/generator-brisket）。

第 15 章

Colony案例研究：脱离Node创建同构应用

Patrick Kunka、Richard Davis、Andrew Barker

Colony 是一个全球性的电影流媒体平台，通过独家的额外内容，它将内容的所有者与热情的粉丝连接到一起。在激烈的市场竞争中，我们的“走出去”战略在很大程度上依赖于世界级的产品和用户体验，以及重新定义电影在线消费方式的雄心壮志。

15.1 问题

Colony 的视频点播模式与 Netflix 这样的竞争对手有一个重要的区别，即我们平台上的内容是开放给公众浏览的，而购买是基于现收现付制的（pay-as-you-go），而不是在订阅、付费之后才能查看。谷歌爬虫会索引我们的整个目录，结果是我们也会因此而获益。在此基础上，我们的产品性质决定了我们需要一个可以经常更新的动态 UI，以反映一个复杂应用的状态变化。例如，元素必须能够更新，以反映用户是否登录、内容包裹（或者其中的一部分）是否属于用户或是否处于租赁期内。虽然原型是基于开箱即用的 ASP.NET MVC 构建的，但我们很快就意识到，使用 SPA 来构建前端界面将能够大大提高我们应对这些挑战的能力。同时，通过技术栈的“解耦”，前后端的团队可以彼此独立地开展工作。

因此，我们面临着进退两难的困境，如何才能将一个传统的、可索引的网站与 SPA 整合起来呢？ 2014 年，Meteor 这样的同构框架已经提供了这种功能的开箱即用，但需要依赖

Node.js 后端。Angular 和 Ember 这样的 SPA 框架已经无处不在，但这些框架都有着 SEO 的缺点，这是我们想要避免的。

由于我们的后端已经在 ASP.NET MVC 构建，且开发团队中有三分之二的人是 C# 开发者。我们想知道，在不依赖 Node.js 以及不需要从零开始重构应用的情况下，是否有方式可以实现同构应用具备的功能。通常来说，我们认为 C# 和 JavaScript 是两种分离且不兼容的技术，如何才能将这两种技术整合到一起？换句话说，是否可以脱离 Node.js 构建同构应用？

解决这个问题将为我们带来许多性能上的优势，并允许我们可以在某个特定的状态中立即渲染应用。例如，链接到一个电影页面之后打开其预告片，或者在特定的时间点渲染结账过程。由于设计中的大部分 UI 是在“模型”中出现的，这些内容传统上只能通过 JavaScript 进行渲染，并且只能在前端管理。

我们想让服务器可以根据 URL 在任何特定的状态下渲染任何页面 / 视图，随后让客户端 JavaScript 应用在后台运行，以接管渲染与路由的工作。为了实现这一点，首先需要一个前后端通用的模板语言以及共享的模板。其次，需要通用的数据结构来表达应用状态。

在此之前，视图模型一直是后端团队负责的，但在理想的解耦的技术栈中，所有的视图设计和模板都应该由前端团队负责，包括传递给模板的数据结构和命名。

从消除所有重复努力或重复代码的角度来看，需要考虑以下几个问题。

(1) 是否有模板语言同时拥有 C# 和 JavaScript 的版本？要想实现这一点，模板语言就需要符合“无逻辑”，不能像 C# Razor 或者 PHP 那样在模板中包含任何的应用代码。
(2) 如果组件需要在前后端间共享，是否可以通过一种与语言无关的数据格式来表达，如 JSON ？
(3) 能否以一种语言编写某些内容（如视图模型），然后自动将其转译为另一种语言呢？

15.2 模板

开始调研模板吧！我们非常喜欢 Handlebars，因为它具有严格的无逻辑生态以及有意限制的作用域。只要给定一个模板和一个任意结构的数据对象，Handlebars 就可以输出一个渲染后的字符串。由于它不是一个完整的视图引擎，因此只可以用它来渲染一些小片段或者将它整合到更大的框架中，以渲染整个应用。其无逻辑的哲学限制了模板逻辑只有 `#if`、`#unless` 和 `#each`，如果需要使用更加复杂的逻辑，那么就不适合使用 Handlebars 了。

虽然最初是为 JavaScript 编写的，但 Handlebars 也可以使用其他语言实现，然后就可以用在其他语言中了。如今，几乎每一种流行的服务器端编程语言都有 Handlebars 的实现。值得庆幸的是，我们在 Handlebars.Net 中找到了维护良好的 C# 版本的开源代码实现。

我们的前端没有使用整个页面模板，而是采用了一种模块化或者“原子类”的设计哲学，任何特定的视图都是由一系列可重复使用的“模块”构成的，模块理论上是可以任意排列并放置在任何上下文当中的。在之前使用 Razor 实现的服务器端视图渲染的解决方案中，这个概念并没有很好地从前端传递到后端，因为后端团队需要拿到静态模块，将它们拼接成更大的结构，并插入到所需的逻辑与模板标签中。然而，拥有共享的模板语言与数据结构之后，就没有理由不在这两个技术栈间共享这些独立的模块了，并且使用 C# 或者 JavaScript 进行渲染会得到相同的结果。这样的话，整个设计模块、模板化和动态化的过程就可以完全交给前端团队来完成了。

15.3　数据

我们之前的解决方案中有一个问题，即原型中的视图模型是临时性的，当添加功能时，需要有机地演变每个模板。之前从未需要获取整个应用的数据结构，因为数据只需要暴露给其中的某一部分视图就可以了，现在两端的团队在创建新的模板时都需要为此而付出代价。

早期为了避免这个问题，我们为整个应用设计了一个“全局性的”数据结构，以便在这个单独的对象中表示站点、资源与应用状态。在后端，这个结构化后的数据会从数据库中映射出来，并传递给 Handlebars 来渲染视图。而在前端，一开始会通过一个 REST API 来接收数据，并将数据传递给 Handlebars。不过，在这两种情况下，提供给 Handlebars 的最终数据都是完全相同的。

我们决定将数据结构设计成如下所示的样子。可以将这个对象认作整个应用任何时候的状态容器：

```
{
  Site: {...},
  Entry: {...},
  User: {...},
  ViewState: {...},
  Module: {...}
}
```

`Site` 对象保存整个网站或应用的相关数据，而不关心当前正在查看的资源。静态文本、Google Analytics ID、任何功能以及环境的切换开关都可以包含在这里。

`Entry` 对象保存了当前被查看资源的相关数据（如页面或特定的电影）。当用户在网站中进行跳转时，被请求的资源的数据从一个特定的端点被拉取，此时就需要更新 `Entry` 属性。

```
<head>
...
  <title>
```

```
    |


    </title>
  </head>
```

User 对象保存了当前登录用户的非敏感数据，如名称、头像和邮箱地址：

```
  <aside class="user-sidebar">
    <h3>Hello !</h4>

        <h4>Your Recent Purchases</h4>

           ...


  </aside>
```

ViewState 对象用于反映视图的当前渲染状态。这是后端根据 URL 来渲染复杂状态的决定性因素。例如，渲染视图时可以选择一个特定的标签、打开模态框或者展开手风琴，只要这个状态拥有自己的 URL：

```
  <nav class="bundle-nav">
    <a href="/extras/" class="tab tab__active">Extras</a>
    <a href="/about/" class="tab tab__active">About</a>
  </nav>
```

Module 对象不是全局性的，但当数据必须传递给某个特定的模块且不能暴露给其他模块时，那么就需要使用这个对象。例如，我们可能需要遍历列表条目中的物品（如一个画廊的图片），逐一渲染图像模块，但其中的数据并不相同。必须使用的数据可以通过 Module 对象直接进行传递，而不是让模块自己作为键的索引从条目中取出：

```
  <figure class="image">
    <img src="" alt=""/>

        <figcaption>
            <p></p>
        </figcaption>

  </figure>
```

15.4　转译视图模型

在定义好顶层的数据结构后，现在需要定义这些对象的内部结构。C# 是强类型的语言，因此不能在松散的 JavaScript 风格中任意地传递这些 C# 的动态对象。每个视图模型都需要严格定义属性和属性各自对应的类型。由于想让前端负责视图模型的设计工作，因此我们决定使用 JavaScript 来编写视图模型。这样做还能够让前端团队的成员可以轻松地测试模板的渲染（例如，使用一个简单的 Express 开发应用），而无须将其整合到 .NET 后端。

JavaScript 构造函数可以用来定义与类相似的对象的结构，其中的默认值可以用于推断类型。

以下是我们应用中一个典型的 JavaScript 视图模型，它描述了一个 `Bundle` 并继承自另一个名为 `Product` 的模型：

```
var Bundle = function() {
  Product.apply(this);
  this.Title    = '';
  this.Director = '';
  this.Synopsis = '';
  this.Artwork  = new Image();
  this.Trailer  = new Video();
  this.Film     = new Video();
  this.Extras   = [new Extra()];
  this.IsNew    = false;
};
```

在显示或隐藏某些特定元素前，模板经常需要检查多个数据块。然而，为了保持模板的简洁，Handlebars 中的 `#if` 语句只能判断一个属性。而且，如果没有自定义的辅助函数，那么也是不能进行比较的，在我们的示例中，代码就会变得重复。虽然更复杂的逻辑可以通过嵌套逻辑语句来实现，但这样创建出的模板可读性差、难以维护，而且违背了无逻辑模板的哲学。我们需要决定在哪里放置这些附加逻辑，从而更加符合我们的情况。

多亏 ES5 JavaScript 提供了“getter”功能，我们可以轻松地向构造函数中添加动态计算的属性值，事实证明，这是最适合进行更复杂运算和比较的场所。

以下代码是视图模型中一个动态的 ES5 `getter` 属性，这个属性会从相同的模型中计算两个其他的属性：

```
var Bundle = function() {
  ...
  this.Trailer = new Video();
  this.IsNew = false;
  ...
  Object.defineProperty(this, 'HasWatchTrailerBadge', {
      get: function() {
          return this.IsNew && this.Trailer !== null;
      }
  });
};
```

现在我们已经定义好了所有的视图模型，其中包括了带类型的属性和 getter。但问题仍然在——它们仅存在于 JavaScript 中。我们想到的第一种方法是，在 C# 中手动重写一遍所有的视图模型，但我们很快意识到这属于重复劳动，而且也不好扩展。我们觉得只要有合适的工具，就可以自动化这个过程。能否通过某种方式将我们的 JavaScript 构造函数“转译”为 C# 中的类呢？

我们决定创建一个简单的 Node.js 应用来完成这项工作。通过使用枚举、类型检查和原型继承，我们可以为每一个构造函数创建描述性的“清单文件”。这些清单中的信息包括类的名称以及继承于哪个类（如果有的话）。在属性的层次，它们则包括了所有属性的名称和类型，以及属性是否为 getter；如果是 getter，则包括 getter 的计算方式是什么，以及返回值的类型。

当每个视图模型被解析为清单文件后，数据就可以放置在 C# 类的 Handlebars 模板中了（似乎有点讽刺）。现在你拥有了一系列可在后端生产环境中使用的的 .cs 文件，每个文件都描述了一个特定的视图模型。

以下是同一个 Bundle 视图模型转译为 C# 的版本：

```
namespace Colony.Website.Models
{
  public class Bundle : Product
  {
      public string Title { get; set; }
      public string Director { get; set; }
      public string Synopsis { get; set; }
      public Image Artwork { get; set }
      public Video Trailer { get; set; }
      public Video Film { get; set; }
      public List<Extra> Extras { get; set; }
      public Boolean IsNew { get; set; }
      public bool HasWatchTrailerBadge
      {
          get
          {
              return this.IsNew && this.Trailer != null;
          }
       }
    }
}
```

值得注意的是，我们的转译器功能有限，只能简单地将一个 JavaScript 构造函数转换为 C# 类。它不能将任意的 JavaScript 代码转换成等价的 C# 代码，这是一项复杂无比的任务。

15.5 布局

我们需要一种定义方式，即需要为某个特定视图渲染哪些模块，以及通过什么方式进行渲染。

如果让视图控制器负责这项工作，那么模块的列表以及任何附带的逻辑都需要在 C# 和 JavaScript 中重复编写。为了避免这种重复，我们想要使用一个 JSON 文件来表示每一个视图（例如，以下的示例描述了首页的一个可能布局），并且该文件可以在前后端之间共享：

```
[
    "Head",
    "Header",
    "Welcome",
    "FilmCollection",
    "SignUpCta",
    "Footer",
    "Foot"
]
```

虽然按照特定顺序列出模块的名称很简单，但我们还想在特定条件下选择性地渲染某些组件。例如，当用户登录时才渲染用户侧边栏。

我们从 Handlebars 有限的逻辑集合（`#if`、`#unless` 和 `#each`）中获得了灵感。我们意识到，通过引用上述提到的全局数据结构中的属性，可以利用 JSON 来表达我们需要的所有内容：

```
[
    ...
    {
        "Name": "UserSidebar",
        "If": ["User.IsSignedIn"]
    },
    {
        "Name": "Modal",
        "If": ["ViewState.IsTrailerOpen"],
        "Import": "Entry.Trailer"
    }
]
```

重组布局的格式后，我们就有能力表达简单的逻辑并将任意数据导入模块。需要注意的是，`if` 语句会接收一个数组，从而允许判断多个属性值。

我们开始进一步研究如何通过这种格式来描述更复杂的视图结构，让模块可以嵌套在其他模块中：

```
[
    ...
    {
        "Name": "TabsNav",
        "Unless": ["Entry.UserAccess.IsLocked"]
    },
    {
        "Name": "TabContainer",
        "Unless": ["Entry.UserAccess.IsLocked"]
        "Modules": [
            {
                "Name": "ExploreTab",
                "If": ["ViewState.IsExploreTabActive"],
                "Modules": [
                    {
                        "Name": "Extra",
```

```
                        "ForEach": "Entry.Extras"
                    }
                ]
            },
            {
                "Name": "AboutTab",
                "If": ["ViewState.IsAboutTabActive"]
            }
        ]
    }
    ...
]
```

这种表示嵌套模块的能力允许我们可以完全自由地实现标记的构建。

现在我们已经拥有了一种强大的格式，可以描述布局和视图的结构，并实现相当多的逻辑。在此之前，无论使用 JavaScript 还是 C# 来编写这个逻辑，都需要繁琐和容易出错的手动复制。

15.6　页面生成器

要想在任何一端获得最终渲染的 HTML，我们需要取得模板、布局和数据，并将它们整合在一起以生成视图。

我们对这部分功能达成了一致的观点，这些功能需要重复，分别具备 C# 和 JavaScript 的版本。我们为一个类设计了规格，并称之为“页面生成器”，这个类负责遍历一个给定的布局文件，按照布局文件中的条件和每个模块各自的数据来渲染模块，最后返回一个被渲染的 HTML 字符串。

因为需要密切关注这两种语言的实现确实返回了相同的输出，所以我们将 C# 和 JavaScript 两个版本的页面生成器都作为分组编程练习，从而让整个开发团队参与进来，这也是让前后端团队成员互相学习对方技术的一个好机会。

15.7　前端SPA

我们开发团队的文化一直是尽可能地构建自研发的技术。当谈到 SPA 时，我们可以选择使用 Angular、Backbone 或 Ember 这样现成的框架，也可以选择构建自己的框架。我们已经处理完模板，且覆盖 API 层的全局数据结构也有效地形成了应用的状态，接下来我们仍然需要一些组件来管理路由、数据绑定和 UI。我们一直坚信“非侵入式的”JavaScript 的概念，因此 Angular 中模糊 HTML 与 JavaScript 界限（包括现在 React 的 JSX 也是如此）这样的用法是我们想要避免的。我们也意识到，如果尝试在非常独特的架构上强行套用一个框架，那么将会导致二次修改，也会使框架出现大量的冗余。因此，我们决定构建自己的解决方案，并且清晰地规定了几个原则：首先，UI 行为不应该和特定的标记紧耦合；其

次，应用状态变化的组合及在模板与布局中已经定义好的 Handlebars 逻辑应该足以使网页上的任何元素在任何时间都能够重新渲染。

我们达成的解决方案不仅非常轻量级，而且非常模块化。在最底层，从服务器端取得状态与完整渲染的视图，在这里我们可以让任意的标记块“订阅”状态的变化。这些变化通过 DOM 事件触发，我们发现这比 Angular 这样的 digest 循环或者各种试验性的“可观测”实现的性能要强大得多。当改变发生时，那部分 DOM 结构会被重新渲染并替换。最近我们放心地发现，其基本原理和越来越流行的 Redux 库几乎是一致的。

往上一层，我们的 UI“行为”和这个过程是完全分离开的，这有效地逐步增强了任意标记块。例如，我们可以将相同的“滑块”UI 行为应用到各种不同的组件中，但每个组件却可以有完全不同的标记，即一种情况可以是电影列表，而另一种可以是新闻引用列表。

在最高层次，History API 通过拦截所有路由点击来提供路由功能，并确定结果视图需要用到哪些布局文件。在通过服务器端 REST API 传递的、形成应用状态的数据的基础之上，我们决定扩展这个 API，以提供模块模板与 JSON 布局。这样做可以确保这些共享资源只存在于一个地方（服务器），从而减少前端与后端资源出现分歧的风险。

15.8　最终架构

图 15-1 的示意图展示了初始的服务器端请求与后续所有的客户端请求的完整生命周期，其中包括它们各自用到的组件。

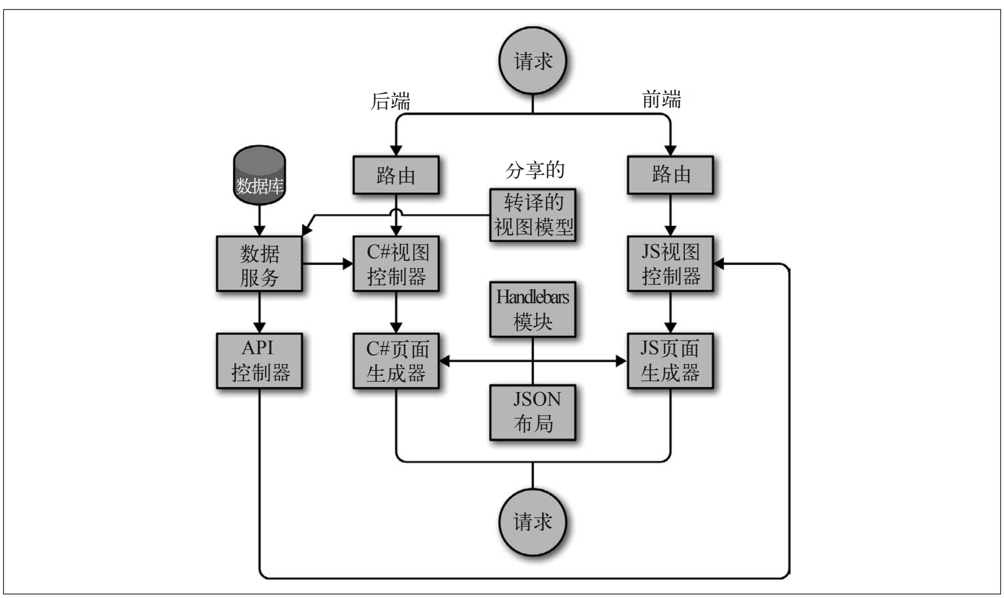


图 15-1：初始的服务器端请求与后续的客户端请求的生命周期

15.9 后续计划

尽管在应用中共享这些不同的组件已经大大减少了重复代码，但依然存在一些重复的情况，比如控制器和路由。然而，根据我们处理布局文件的经验，为什么路由与各自对应的控制器不能通过共享 JSON 文件的方式来表达，然后交给轻量级的解释器进行解析，从而得到 C# 和 JavaScript 版本的实现呢？

虽然一些解决方案已经实现了使用非 JavaScript 引擎（如 ReactJS.NET 和 React-PHP-V8JS）来渲染 Angular 应用或 React 组件，但因为 ASP.NET 生态系统缺乏特定的依赖框架，所以我们的解决方案依然是唯一的。在解决方案的开发过程中，虽然我们的后端团队确实编写了一些附加代码（如页面生成器），但他们的数据服务、控制器以及应用逻辑或多或少是未有变化的。由于解耦，前端 UI 代码可以从后端中分离出来，从而让后端应用变得精简而目标明确。虽然我们的应用在某种意义上是“同构的”，但其同时也是划分整齐、关注点分离的。

我们猜测还有很多其他的开发团队像我们一样渴望着同构应用带来的优势，同时又受困于后端技术的选择——无论是 C#、Java、Ruby 还是 PHP。在这些情况下，公司早期作出的决定很可能会对产品架构造成根深蒂固的影响，并影响开发团队的组成结构。我们的解决方案表明，同构的原则可以适用于任何技术栈，既不需要引入一个可能在一年内就被淘汰的前端框架上的依赖，也无须从头开始重构整个应用。希望其他团队可以从我们的经验中获得灵感。我们期待可以在未来看到一系列的技术加入到同构应用中。

第16章
结语

Charlie Robbins

> “我不知道世人如何看待我，但我觉得自己像是在海边玩耍的孩子，只是偶尔拾到了一枚比较光滑的卵石或一只比较好看的贝壳，而展现在我面前的是完全未被探明的真理之海。”
>
> ——艾萨克·牛顿

2009 年 10 月前，我还没有专业地编写过 JavaScript。准确来说，我几乎没花过多少时间在这门语言上。当时我主要是使用微软的 WPF（Windows Presentation Foundation）技术编写交易系统的前端界面以及其他的银行软件。如果你在那时告诉我，“同构 JavaScript”将会成为一种广泛使用的概念，我肯定会嗤之以鼻。通过强调这些事实，我想提醒你的是，在进行尝试以前，你真的不知道未来可能发生什么。

16.1　设计模式、Flux和同构JavaScript家族

在我原本的博文“Scaling Isomorphic Javascript Code”（https://blog.nodejitsu.com/scaling-isomorphic-javascript-code/）中，同构 JavaScript 并不是主要的话题。在那时，这个术语仅仅是一个脚注说明，为的是证明我的想法是一个更加有影响力的软件设计模式。开发者采用术语和抽象概念的方式远远超过了任何具体的软件设计模式。

软件设计模式的发展方式有点类似于宗教：每种模式的核心思想都有多种解释，这可能会导致各教派的特征千差万别。当然，在软件设计模式中，一个教派实际上只是模式的另一

种具体实现。我将这种原则称为“设计模式家族”原则。我们以 MVC 模式（Model-View-Controller，模型－视图－控制器）为例，它可以说是一直以来最流行的设计模式了。MVC 的原始实现诞生于施乐帕克研究中心的 SmallTalk 程序语言，它与如今 Ruby on Rails 中类似于 Model2 的 MVC 实现很少有相似之处。这些差异主要是因为 Ruby on Rails 是一个基于服务器的框架，而原始的 MVC 实现的关注点是前端的桌面图形用户界面。

面对日益增长的同构 JavaScript，设计模式与软件架构都需要发展。实践表明，自从 Flux 模式家族出现，同构 JavaScript 和单向不可变数据流的这种转变验证了“设计模式家族”的原则。Flux 产生了大量的实现，其中大部分的实现是同构的。这些实现的普及和它们同时支持客户端与服务器端渲染是相关的。React、Flux 以及 Redux 这样的流行实现的兴起验证了同构 JavaScript 的重要性。事实上，Redux 的作者关于这个问题有着自己的看法（https://medium.com/@mjackson/universal-javascript-4761051b7ae9#.h655sp39b）。

16.1.1　永远相信JavaScript

自 2011 年 10 月起，在我第一次提出同构 JavaScript 后，JavaScript 就遍地开花了。如果将 `npm` 模块的数量作为增长的指标，那么从 2011 年年末到 2016 年年中，JavaScript 取得了高达 50 **倍**的增长。回过头来看，这是一个惊人的数据，更为惊人的是，JavaScript 的增长并没有显现出放缓的迹象。

JavaScript 的创造者 Brendan Eich 曾经讲过一句经典的话：**永远相信 JavaScript**。现在看来确实如此。JavaScript 在今天几乎无处不在，为不同的平台、设备及终端提供支持。JavaScript 在手机、无人机和汽车中运行，还帮助 NASA 跟踪宇航服，而我现在使用的文本编辑器也是使用 JavaScript 编写的。

比 JavaScript 的使用率发展更为迅猛的是 JavaScript 的开发方式。2011 年，第一个 JS-to-JS 式的编译器（或称为转译器）Traceur 由 Alex Russell 在 JSConf（https://www.youtube.com/watch?v=ntDZa7ekFEA&t=1m42s）发布，并被称为“一个有有效期的工具”。Traceur 和 `npm`、`browserify` 和 `webpack` 这样的 JavaScript 工具让转译功能得到了广泛使用，且使用方法比之前简单多了。编写本章时恰逢 ES6/ES2015 的规范定稿。

这些因素结合起来奠定了现代 JavaScript 开发的基石。其思想是，在浏览器执行 JavaScript 代码前，**总是先轻松地运行 babel 这样的转译器**，并且以一种快速、渐进的方式采用 ES201{5,6,7...} 的功能。

当然了，“总是”是离不开“轻松”的，而变得轻松要经历一段发展。采用 `npm` 作为工作流的工具可以使得管理 JavaScript 语言工具更为轻松。当必须在 `react` 和 JSX 这样的框架中使用转译时，拥有易于使用的语言工具将会使得“使用转译器”这一想法变得没那么令人厌恶。

这就为同构 JavaScript 创造了一个令人难以置信的友好环境。考虑以下场景：当一个库不

能完全适合你的应用环境或者语义时，在这些普遍存在的工具链出现之前，你是不能使用这个库的，但现在你的转译和打包的工具链通常可以帮助你将这个要求变为可能。

其他一些激动人心的特性即将出现。WebAssembly（简称 `wasm`）是一项适用于 Web 编译的、可移植的、体积紧凑且加载高效的格式。它为许多语言创造了一个可行的编译目标，更重要的是，在那些语言中建立了良好的项目。WebAssembly 承诺的愿景将会超越同构 JavaScript，为通用的同构代码提供看似无尽的可能性。

16.1.2 命名与理解

最后，我想讨论一下我经常被开发者问到的几个问题。第一个问题是，为什么要构造“同构”这个新术语？这个新术语的含义其实十分明显，就是在客户端与服务器端运行同一份代码。我给出的答案很简单：因为还没有一个专门的词来描述它。还有人问我，为什么这个词是源于一个数学概念？虽然我知道自己的答案多少有些争议性，但对我来说却是显然的：因为前端的构建系统以及最近以来的转译器代表了我认为的“同构”理念。

> iso·mor·phic（形容词）
> (1)
> a. 形式、形状和结构相等或相似
> b. 在规模和形状中可能存在相似的孢子体和配子体世代
> (2) **和同构相关的**

最近，我更频繁地被问及“同构 JavaScript”与“通用 JavaScript”之间的辩论。要想客观地探讨这一问题，则需要再次回顾本书的第 2 章与第 3 章。同构 JavaScript 图谱展示了在开发与构建同构 JavaScript 应用时复杂程度的分类。同构 JavaScript 三个不同的分类展示了依据复杂度与环境来调整 JavaScript 的三个等级。

- **与环境无关的** JavaScript 可以“到处”运行，无须修改。
- 只有当环境本身被修改后，为每个环境提供 shim 的 JavaScript 才能在不同的环境中运行。
- 使用 shim 后语义的 JavaScript 可能需要修改环境或者在不同环境中的行为略有不同。

如果“通用 JavaScript”指的是不需要修改代码或环境就可以实现重构，那么**与环境无关的** JavaScript 显然是通用的。因此，当需要修改代码本身或运行环境时，代码就不再是通用的了，而更加趋向于同构化（如图 16-1 所示）。



图 16-1：通用与同构 JavaScript 图谱

当我们将图谱和分类结合起来，问题就从“这是同构 JavaScript 还是通用 JavaScript？”变成了“这段 JavaScript 是更偏向于同构化还是更偏向于通用化？”这种解释比我原来对同构 JavaScript 的定义更有技术性，也更为细致：

> 同构意味着给定任意行数的代码（除了某些特例）都可以在客户端和服务器端执行。

这也说明了一个关键点，即对这些术语的辩论和解释是无止境的，但这在很大程度上来说却是不重要的。例如，让无处不在的 JavaScript 开发者感到惧怕的是，我可以提出自同构（automorphism）这一术语，它指的是“源和目标一致的同构”。根据这一点，我们可以推断出“自同构 JavaScript”是“通用 JavaScript”的另一种说法，因为其代码在所有环境中都是相同的。这个新术语不会对这一主题的共同理解产生任何实际价值，只会进一步混淆一个简单（尽管是微妙的）的想法。

在这时候，你可能会问自己，“技术上的细微差别对我们开发者来说不是很重要吗？”虽然技术开发中总是不会缺乏细微的差别，但开发者不应该将关注点放在一个新术语或者不同术语的争论上。开发者应当将重点放在利用自己的理解（无论对错）来做一些有趣或创新的事情。我们对于所有事情的共同理解都是在不断变化的，其中包括 JavaScript。所有的代码城堡最终都将被淘汰，而我则期待看着它们坠入大海。

关于作者

Jason Strimpel 是 WalmartLabs 平台团队的一名资深软件工程师，专门从事 UI 层的开发。Jason 拥有 12 年构建 Web 应用的经验。大约在三年前，他开始专注于前端领域，尤其是 JavaScript。从那时起，他开始和一些组件库及框架打交道。然而，在提出一些独特的、具有挑战性的 UI 需求时，Jason 发现这些库具有局限性，因此他开始开发自己的定制组件与辅助工具目录。他是一个极具热情的开发者，但缺乏幽默感，喜欢在构建丰富的 UI 时简化复杂度。

Maxime Najim 是一名软件架构师，同时也是一名全栈 Web 开发者。他曾任职于 Yahoo!、苹果和 Netflix，在创建大型的、伸缩性强的、可靠的 Web 应用方面具有丰富的经验。目前，他正专注于设计并实现 Walmart 全球电商平台的新系统与新框架。

关于封面

本书封面上的动物是灰树蛙（Hyla versicolor），是北美洲中东部的一种小型蛙类。灰树蛙因在树木栖息且身体的主要颜色为灰色而得名。不过，正如它的学名所示，灰树蛙可以从灰色变为绿色，就像变色龙一样。这种伪装能力可以用于躲避天敌。

灰树蛙的分布范围非常广泛，在美国境内，南至得克萨斯州，北至新不伦瑞克，均可找到它们。除了交配时，灰树蛙一般很少离开树上的家。灰树蛙体长约两英寸（约 5 厘米），皮肤粗糙，表面有疣，与蟾蜍类似（因此得名为“多变的蟾蜍”）。主要的食物为昆虫，如蟋蟀、蚱蜢和蚂蚁。

灰树蛙的生存没有受到什么重大的威胁，种群数量被认为是稳定的。灰树蛙有时会被当作宠物饲养，寿命为 5~10 年。

O’Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

封面图片来自 Wood 的著作 *Illustrated Natural History*。

O'REILLY®

同构JavaScript应用开发

被誉为“Web开发圣杯”的同构JavaScript正在改变着Web应用的开发模式，实现在浏览器客户端和Web应用服务器端运行相同的代码。通过阅读本书，你将逐步了解这种应用架构日益流行的原因，学会如何构建和维护属于自己的同构JavaScript应用，并将其应用于解决关键业务问题。

- **了解同构JavaScript如何显著提升用户体验**
- **定义框架和应用之间的合约，响应资源请求**
- **将框架和应用代码从服务器端传递到客户端，让代码库变为同构**
- **创建常用抽象，获取和设置cookie，重定向用户请求**
- **了解同构JavaScript为何终将解决富服务器端和富客户端之争**
- **同构JavaScript的高级话题，如协作性的实时应用**

Jason Strimpel，软件工程师，拥有十余年Web开发经验。目前任职于沃尔玛实验室，负责支持UI应用的软件开发。

Maxime Najim，沃尔玛实验室软件架构师，全栈Web开发者。曾任职于Netflix、苹果和Yahoo!等公司，在创建大型、伸缩性强、可靠的Web应用方面具有丰富经验。

“同构JavaScript应用在追求速度的场景中表现优异。对于任何想要构建新型高性能Web应用的人来说，这本书都值得一读。”

——Alexander Grigoryan
沃尔玛全球电子商务应用
平台软件工程总监

“本书内容详实、结构清晰、讲解清楚。作者以新鲜的视角阐述了Web应用架构的沿革，详细解读了为何同构JavaScript是Web应用和单页面应用架构的完美结合，以及它区别于简单地在客户端和服务器端分享代码的独到之处。”

——Amazon读者

JAVASCRIPT

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 软件开发 / JavaScript

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-46868-0

ISBN 978-7-115-46868-0
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks